FORMALIZING REQUIREMENTS IN A COMMERCIAL SETTING: A CASE STUDY

by

André Wong

A thesis submitted in conformity with the requirements for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 1999 by André Wong
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.
Abstract

Formalizing Requirements in a Commercial Setting: A Case Study

André Wong
Master of Science
Graduate Department of Computer Science
University of Toronto
1999

Formal description techniques (FDTs) have long been advocated as a solution to problems posed by requirements specified in natural language; however, successful applications of FDTs have been confined mostly to safety-critical projects.

This thesis describes a research project conducted in collaboration with Nortel to investigate the technical and the economic feasibility of applying FDTs in a commercial setting. By first interviewing the Nortel engineers, we identified problems with their development process and proposed criteria for our quantitative formal description method evaluation. We selected a method that was suitable to our project and usable in the Nortel environment and applied it to a multimedia-messaging subsystem; the resulting model was used to identify requirements errors and to derive test suites, shadowing the existing development process and keeping track of a variety of productivity data. By amortizing the cost of formalization to the testing phase, we showed that the use of FDTs could increase the quality of the software without lengthening the development cycle.
Acknowledgements

Thanks to my supervisor Dr. Marsha Chechik for her inspiration, advice, assistance, and patience throughout my graduate career and, in particular, during the development of this thesis. I would have never completed it without her. I would also like to thank Dr. John Mylopoulos for providing invaluable comments about this thesis, and Dr. Dave Wortman and Dr. Rich Hehner for reading the drafts of the interview scripts.

Thanks to the Nortel engineers who participated in the interview study for devoting their precious time and sharing their knowledge. I can only acknowledge them anonymously because of standard ethics procedures. I am also indebted to Avinash Persaud and Shawn Turner of Nortel Networks who made this joint project possible.

Thanks to my Mom, May Wong, for the unfailing emotional support and encouragement throughout the years and to my girlfriend, Carmen Tang, for the extraordinary patience and understanding for the past five years.

Last, thanks to Ontario Graduate Scholarship, the Department of Computer Science at the University of Toronto, and Nortel Networks for the grateful financial support.
Contents

1 Introduction
  1.1 Formal Description Techniques .............................................. 1
  1.2 Industrial Applications ...................................................... 3
  1.3 Obstacles to Applying Formal Descriptions Techniques to Commercial Systems ...................................................... 4
  1.4 Tailoring Formal Description Techniques .................................. 5
  1.5 Formal Description Techniques Studies ................................... 7
  1.6 The Origin of the Project .................................................... 8
  1.7 Project Overview .......................................................... 8

2 A Study of a Commercial Software Development Process .................. 11
  2.1 Overview .............................................................................. 11
  2.2 Nortel Software Development Process ..................................... 12
  2.3 Method .................................................................................. 13
    2.3.1 Defining the Population .................................................. 14
    2.3.2 Formulating the Interview Questions ................................ 14
    2.3.3 Conducting the Interviews ............................................... 15
    2.3.4 Qualitative Data Analysis ............................................... 16
    2.3.5 Threats to Validity ......................................................... 17
  2.4 Results .................................................................................. 18
    2.4.1 Software Requirements Documents .................................... 18
    2.4.2 Manual Inspections ......................................................... 21
    2.4.3 Testing ........................................................................... 23
    2.4.4 Propagation of Specification Updates ............................... 25
  2.5 Applying the Results ........................................................... 27
  2.6 Summary .............................................................................. 29

3 Software System Selection .......................................................... 30
  3.1 Overview .............................................................................. 30
  3.2 System Selection ................................................................... 30
  3.3 System Description .............................................................. 31
    3.3.1 PasswordCheck .............................................................. 33
    3.3.2 ThruDial ........................................................................ 35
  3.4 Summary .............................................................................. 38
# 4 Formal Description Method Selection

4.1 Overview ................................................................. 40
4.2 Rationale for Using Formal Description Methods .............. 40
4.3 Preliminary Screening .................................................. 42
4.4 Method ........................................................................ 43
4.5 Quantitative Feature Analysis ........................................ 43
   4.5.1 Criteria ............................................................... 44
   4.5.2 Limitations .......................................................... 51
4.6 Evaluating Formal Description Methods ............................ 51
   4.6.1 SDT .................................................................. 52
   4.6.2 TestMaster ......................................................... 55
   4.6.3 Validator/Req ...................................................... 58
   4.6.4 Evaluation .......................................................... 61
4.7 Summary ....................................................................... 62

# 5 Formal Requirements Modeling

5.1 Overview ...................................................................... 70
5.2 Method ......................................................................... 70
5.3 Formal Modeling .......................................................... 72
   5.3.1 Naming Conventions ............................................ 73
   5.3.2 A ServiceCreator Application ............................... 73
   5.3.3 ThruDial ............................................................. 81
   5.3.4 Level of Abstraction ............................................ 94
   5.3.5 Test Case Derivation ............................................ 95
   5.3.6 Specification and Implementation Problems in ThruDial .. 102
5.4 Findings ...................................................................... 106
   5.4.1 Quantitative Results .......................................... 107
   5.4.2 Qualitative Observations ..................................... 109
   5.4.3 Limitations ........................................................ 111
5.5 Summary ...................................................................... 111

# 6 Usability Workshop

6.1 Overview ..................................................................... 113
6.2 Description .................................................................. 113
6.3 Modeling Exercises ..................................................... 114
6.4 Questionnaire ............................................................. 116
6.5 Summary ..................................................................... 117

# 7 Summary and Future Work

7.1 Lessons Learned .......................................................... 119
7.2 Limitations ................................................................. 120
7.3 Improvements and Future Work .................................... 121

Bibliography .................................................................... 123
### A Interview Question Sets

<table>
<thead>
<tr>
<th>Question Set</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Question Set for Interviewing Software Designers</td>
<td>134</td>
</tr>
<tr>
<td>A.2 Question Set for Interviewing Software Testers</td>
<td>139</td>
</tr>
</tbody>
</table>

### B ServiceCreator Components: Original Specifications

<table>
<thead>
<tr>
<th>Component</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 PasswordCheck</td>
<td>144</td>
</tr>
<tr>
<td>B.2 ThruDial</td>
<td>148</td>
</tr>
</tbody>
</table>

### C SDL: A Brief Description

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1 Background</td>
<td>157</td>
</tr>
<tr>
<td>C.2 Overview</td>
<td>157</td>
</tr>
<tr>
<td>C.2.1 Systems, Blocks, and Processes</td>
<td>159</td>
</tr>
<tr>
<td>C.2.2 Other Features</td>
<td>161</td>
</tr>
</tbody>
</table>

### D PasswordCheck Models

<table>
<thead>
<tr>
<th>Model/Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1 SDL Model/MSCs</td>
<td>164</td>
</tr>
<tr>
<td>D.2 TestMaster Model</td>
<td>168</td>
</tr>
<tr>
<td>D.3 Use Case Diagrams/Sequence Diagrams</td>
<td>172</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Characteristics of the interviewees. ........................................ 14
2.2 Topics of questions in the second half of question sets QS1 and QS2  15
2.3 Some high-level constructs used in analyzing the interview responses. 16
2.4 A sample data matrix. ..................................................... 17
2.5 Opinions about the usefulness of a high-quality software requirements specification in the various development phases. .................... 18
2.6 Opinions about the major problems in understanding the SRD. .......... 19
2.7 Some possible causes of the problems in the SRDs. ...................... 20
2.8 Opinions about problems in the manual inspection process. ............. 22
2.9 Opinions about problems in creating high-quality test cases. ........... 23
2.10 Responses to tester-specific statements. ................................ 24

3.1 The 15 ServiceCreator components analyzed in our study. .......... 33

4.1 Formal description method evaluation criteria. .......................... 47
4.2 The 6-point scale for rating the degree a method satisfies a criterion. 61
4.3 Evaluation of SDT's notation (SDL). .................................... 63
4.4 Evaluation of SDT. ....................................................... 64
4.5 Evaluation of TestMaster's notation. .................................... 65
4.6 Evaluation of TestMaster. ................................................. 66
4.7 Evaluation of Validator/Req's notation (a subset of UML). ............... 67
4.8 Evaluation of Validator/Req. ............................................. 68
4.9 The scale for rating the relative importance of the evaluation criteria. 68
4.10 Comparison of the three evaluated methods. ............................ 69

5.1 Descriptions of the SDL signals. ......................................... 80
5.2 Productivity comparisons. .................................................. 107

6.1 Characteristics of the participants. ....................................... 114
6.2 Some results from the questionnaire. .................................... 116

C.1 Some SDL symbols. ...................................................... 162

D.1 Some test cases generated for the sequence diagram invalidPW. ........ 175
List of Figures

1.1 Major components of the research project. ........................................... 10
2.1 A simplified view of the development process at Nortel. ...................... 13
2.2 The relationship between the SRD, design, implementation, and test cases. 26
2.3 A vicious cycle. .................................................................................. 27
3.1 A mock-up view of ServiceCreator. ..................................................... 32
3.2 The workspace view of PasswordCheck. ................................................. 34
3.3 The property dialog of PasswordCheck. ............................................... 35
3.4 The property dialog of ThruDial. .......................................................... 36
3.5 The workspace view of ThruDial. .......................................................... 37
4.1 SDL layers and the possible communications channels. ......................... 53
4.2 The hierarchical structure of an FSM in TestMaster. ............................ 56
5.1 The formalization process and the existing development process. .......... 71
5.2 A high-level overview of various constructs in the SDL model. ............. 74
5.3 An SDL system. ................................................................................... 75
5.4 Signal declarations. .............................................................................. 76
5.5 SDL data type declarations. .................................................................. 76
5.6 An SDL block. ..................................................................................... 77
5.7 The SDL process main. ....................................................................... 79
5.8 The graphical view of the ServiceCreator application illustrated in Figure 5.7. 79
5.9 The definition of the constant TD_GTG_NAME. .................................... 81
5.10 Page 1 of the SDL process ThruDial. .................................................... 85
5.11 Page 2 of the SDL process ThruDial. .................................................... 86
5.12 Page 3 of the SDL process ThruDial. .................................................... 87
5.13 Page 4 of the SDL process ThruDial. .................................................... 88
5.14 The procedure fillLeftPadding inside the SDL process ThruDial. .......... 89
5.15 Page 5 of the SDL process ThruDial. .................................................... 90
5.16 Page 6 of the SDL process ThruDial. .................................................... 91
5.17 Page 7 of the SDL process ThruDial. .................................................... 92
5.18 Page 8 of the SDL process ThruDial. .................................................... 93
5.19 Two possible ways to model the action of playing a voice prompt: a) a simplified approach used in the study, and b) a more accurate approach. 94
5.20 The graphical view of the application created for testing ThruDial. ....... 96
5.21 The MSC created for test case #1. .................................................. 100
5.22 The graphical view of the application created for test case #23. ........ 101
5.23 The MSC created for test case #23. .................................................. 101
5.24 A Nortel test case created for testing ThruDial. ............................... 108

6.1 The natural-language description of the first software system. .......... 115
6.2 The natural-language description of the second software system. .... 115

B.1 Workspace view of PasswordCheck block. ....................................... 145
B.2 PasswordCheck .................................................................................. 146
B.3 Workspace view of ThruDial block. .................................................... 148
B.4 ThruDial .............................................................................................. 149

C.1 SDL layers and channels. ................................................................. 158
C.2 The description of a resource manager. .............................................. 158
C.3 The SDL system of the resource manager. ........................................ 159
C.4 The SDL block of the resource manager. .......................................... 160
C.5 The SDL process for the resource manager. ...................................... 161
C.6 An SDL data type example. ............................................................... 163
C.7 An SDL timer example. ..................................................................... 163

D.1 Commonly used SDL symbols ............................................................. 165
D.2 Commonly used MSC symbols ............................................................ 165
D.3 The first page of the SDL model created for the simplified PasswordCheck .......... 166
D.4 An MSC created for the simplified PasswordCheck. .......................... 167
D.5 The first-level FSM of a TestMaster model. ....................................... 168
D.6 Commonly used symbols in the TestMaster language. ...................... 168
D.7 A list of possible passwords for testing PasswordCheck. ................... 169
D.8 The FSM representing the validatePW model. .................................. 170
D.9 A description of a transition in a TestMaster model. .......................... 171
D.10 Two TestMaster test scripts generated from the PasswordCheck model. 171
D.11 The use case diagram for the simplified PasswordCheck. ................. 172
D.12 A sequence diagram describing the interactions that take place when a caller enters an invalid password. ............................................. 173
D.13 A sequence diagram describing the interactions that take place when a caller enters a valid password ................................................. 173
D.14 A sequence diagram describing the interactions that take place when a caller stays idle ............................................................ 174
Chapter 1

Introduction

The commercial software industry has been facing a software development dilemma: the public expects software systems to have the same high quality as found in other engineering artifacts, while the competitive market pressures the industry to place less emphasis on quality and deliver products in the shortest possible timeframe. On the other hand, development techniques used in industry have not kept up with the changing environment; many of them are not suitable for building high-quality complex systems quickly yet economically. Indeed, projects often either are late, are over budget, or deliver systems that do not meet customers’ requirements [Gen91]. Perceiving that high-quality software artifacts are an important part of the solution, some researchers in the software engineering community have advocated the use of precise documentation in the development process to increase the quality of the systems [Par93]. This allows to discover problems earlier, reduce the cost of errors, and shorten the development cycle in the long run.

1.1 Formal Description Techniques

Formal description techniques (FDTs) are mathematically-based languages, techniques, or tools for specifying computer systems [Lar95]. They allow system artifacts, such as system requirements and designs, to be described in an unambiguous mathematical notation. Their use in a project can assume various forms, ranging from occasional mathematical expression in a natural-language specification to a complete system specification

\(^1\)FDTs comprise one component of formal methods, which can be used in the specification, analysis, and verification of software systems [CW96].
using a formal language with precise syntax and semantics. In addition, they are useful in many stages of the development process, from capturing customers' intentions and specifying system-level specifications to designing [AG94, Bow88, Cat95, CG95], implementing [CS98, HK97], and testing [BB97, BPBM97, Don97, TPvB96] software systems. However, since errors discovered late in the development process are much more expensive to rectify [Boe81, Can91, KSH92], and they are usually introduced during the requirements phase [HCK95, Lut93, oD84], more leverage can be obtained by applying FDTs to system requirements early in the development process.

Compared to the traditional way of specifying requirements in a natural language, the use of a formal notation allows the system behavior to be described precisely. Some of the advantages of applying FDTs are as follows [Che96, NS89, Sai96]:

- The process of formalizing specifications provides a more detailed understanding of the systems by encouraging software engineers to analyze the requirements in depth, raising issues that may not otherwise be manifested until later in development;
- Applying FDTs exposes omissions, unstated assumptions and ambiguity that are commonly found in natural-language specifications;
- FDTs help detect requirements errors early in the software lifecycle, providing a potential to reduce the length of the overall development process and the cost of fixing the late errors;
- More precise and cleaner requirements help achieve lower code complexity [PH97].

A large variety of formal notations have been developed for different domains, based on a belief that no single notation can adequately meet all their needs. For instance, state-based notations, such as Specification and Description Language (SDL) [CCI88], Requirements State Machine Language (RSML) [LHHR94], Statecharts [Har87] and Software Cost Reduction (SCR) [Hen78, Hen80], are suitable for specifying reactive systems in which operations can be best described by a series of discrete state changes. The behavior of concurrent or distributed systems can be expressed using process algebras such as Communicating Sequential Processes (CSP) [Hoa85] and Calculus of Communicating Systems (CCS) [Mil89]. On the other hand, more data-centered programs can be represented using abstract data-type notations such as Z [SKW87] or Vienna Development Method (VDM) [Jon90]. Also, by using real-time logic-based notations such as linear-time logic (for describing particular system executions) [MP92] and branching-time logic (for describing all possible system executions) [CES86], it is possible to describe the
temporal behavior by defining a set of properties that a system should satisfy, without constructing the actual model. System requirements and a set of desirable properties can also be expressed in first- or higher-order logics, and inductive techniques, some with tool support (e.g., Larch Prover [GH93] and PVS [OSR93], respectively), can be used to determine that the system, together with the assumptions made about it and its environment, satisfies the properties.

1.2 Industrial Applications

Contrary to the general belief [Hal90], FDTs have been successfully applied to specify industrial systems, and some of these projects are summarized in [CGR93] and [CW96]. Most applications of FDTs involve safety-critical systems. For instance, SCR has been applied to the requirements specifications of the shutdown system in Darlington Nuclear Generation Station [AH+90], the Operational Flight Control Program of the A-7 military aircraft [Hen80], and the weapons control panel of a U.S. military system [HKL98]. NASA applied PVS [OSR93] to the development of Cassini spacecraft's system-level fault-protection software [LA94], the highest-level requirements for Failure Detection, Isolation, and Recovery (FDIR) of the International Space Station [HCK95], the integration of the Global Position System with the existing Space Shuttle software [CV98]. RSML was used to specify the Traffic Collision Avoidance System (TCAS) II [MN96]. Paris Metro used Hoare assertions augmented with the B-Method [ALN+91] in building a software signaling system that controls the train separation [GH90].

Formal methods have also been used to analyze new and existing protocols. The payment properties of major electronic commerce protocols such as Secure Sockets Layer (SSL) and Secure Electronic Transaction (SET) [Mas96] were specified in Murphi [Dil96b] by Mitchell et al [MSS98] and in mathematical logic by Bolignano [Bol97], respectively. Murphi and SMV [McM93] were used in analyzing the cache coherence protocols of the IEEE Futurebus+ (Standard 896.1-199) [CGH+95] and Scalable Coherent Interface (Standard 1596-1992) [DDHY92].

The use of FDTs is not limited to the software systems. In fact, they have been applied more extensively in commercial microprocessor design due to better technical and

---

2Systems where compelling evidence is required that the system delivers its services in manner that satisfies certain critical properties [HBGL95].
CHAPTER 1. INTRODUCTION

When used in various software systems, formal descriptions can provide significant advantages [Dil96a]. For example, Inmos, a British semiconductor manufacturer, used Z and Occam [RH86] to design a scheduler for the T800 transputer\(^3\) and CSP and CCS to specify the design of the Virtual Channel Processor feature in the T9000 transputer [Bar95]. Computational Logic, Advanced Micro Devices, and the University of Texas applied A Computational Logic Applicative Common Lisp (ACL2) [MK97] to formalize the microcode for the floating point square root [Rus97] and division [MTL98] in the AMD\(5_{\times}86\) processor.

Comparatively, there are much fewer large-scale applications of FDTs in commercial software systems\(^4\). The method Z seems to be predominant choice among the more popular case studies; it was used by a consulting firm called Praxis to specify the infrastructure of a Structured Systems Analysis and Design Method (SSADM) [DCC92] tool set [Bro89], by IBM to assist a re-engineer effort of a transaction-processing system [Hoa95, HK91], and by Tektronix to create an abstract specification for the system operations and the user interfaces of a family of oscilloscopes. B-method was used to develop a web-based banking application [Buc98], and SDL has been used by telecommunications companies such as British Telecom [Coo95] and Alcatel [MP98] for building non-safety critical telecommunications systems.

1.3 Obstacles to Applying Formal Descriptions Techniques to Commercial Systems

Although FDTs have been a topic of research for more than 20 years, successes are confined to mostly safety critical systems, which represent only a small portion of the entire software industry—FDTs have yet made real inroad to the much larger commercial software sector. Researchers have attributed the limited adoption to the widening gap between the needs of the commercial software industry (the industry) and what the academia offers [BCKM97, BS93, WW93, MGB+98]. Indeed, a large majority of the research focus solely on increasing software quality, and achieving absolute software quality in the case of formal methods, while the economic aspect of FDTs is largely

\(^3\)A microprocessor that has the processor unit, memory and communications channels integrated in one transputer chip, commonly used in parallel processing systems.

\(^4\)For the purpose of this thesis, commercial systems are defined as large software systems that do not require the level of assurance found in safety-critical systems and are developed by profit-seeking organizations for commercial purposes.
neglected [BB76, Toc97]. Unfortunately, among the few exceptions that we found, Brookes [BFL96] and Pfleeger [PH97] focus on the quantitative benefits of applying FDTs to safety-critical systems over traditional development techniques. The most relevant study is Bicarregui's [BDW96] use of FDTs in developing a part of a commercial application integration product (commonly know as “middleware” in business terms). This study shows that FDTs help detect requirements errors earlier, which indirectly implying that long-term cost savings should result.

On the other hand, profit-driven software companies concentrate on attaining shorter time-to-market, reduced costs, and higher productivity [HB96], and there is a greater emphasis on obtaining immediate and visible benefits than long-term advantages. Often, absolute software quality is a desirable but not crucial objective, and is preferable only if it improves the cost-effectiveness of the development process. Besides, unlike safety-critical systems, most commercial systems do not require the rigorous analyses advocated by most traditional research, and the associated costs are often unjustified. The accompanied assumptions, such as stable requirements and access to experts with deep mathematical background, do not always apply in commercial settings. Practical issues such as usability and maintainability are seldom addressed.

FDTs tend to lengthen the requirements analysis phase. However, commercial software companies are always under pressure to deliver concrete results, e.g. prototypes or working code. Without substantial evidence of the promised benefits from relevant case studies, it is inconceivable that the industry would adapt a relatively radical technique with myths such as requiring huge up-front investments and having incomprehensible notations [Hal90]. In a 1989 survey of practitioners in the telecommunications industry where FDTs have had more successes, 84% and 50% of the subjects believed that FDTs were not viable and would not be viable in three to five years, respectively [NS89]. It is difficult to envision a wider adoption of FDTs until there is a cultural shift in the research community [MGB+98].

1.4 Tailoring Formal Description Techniques

Still, from the overwhelming popularity of Unified Modeling language (UML) [BJR97a, BJR97b], it is evident that there is an unfilled demand for techniques that provide a better understanding of software systems over the traditional specification techniques. Although UML is relatively young and its development is still ongoing, its small up-
front cost, flexibility, widely understood (relative to FDTs) and standardized notation, and industrial tool support have almost made it the de facto modeling language in industry. The key to a wider adoption is to tailor FDTs and their usage to mirror some of UML's successes, which include meeting the most important needs of the industry. The answer does not seem to lie in the increased quality alone but also in the economic benefits [Dil96a] and the ease of adaptation.

**Economic Benefits**

In the profit-driven industry, the motivation for attaining better software quality is to obtain economic benefits. The long-term advantages include lower maintenance cost or increased revenue due to higher customer satisfaction; however, the most crucial return-of-investment is the immediate reduction in development timeframe or cost for increasing the competitiveness. The major challenge is to find ways to obtain short-term savings to compensate for this cost, and one possible way is to leverage the investment to other stages of the software lifecycle, i.e., generating code or test cases from the formal specifications. Not only does this amortize the cost of creating the specifications, but the productivity improvements also become more quantifiable and visible. Still, the total decrease in the development cycle is only achievable if the formalization can be done fairly inexpensively.

**Ease of Adaptation**

The user-friendliness of FDTs correlates strongly with their effectiveness in commercial projects and their acceptance among the practitioners. As most practitioners do not possess a deep mathematical background, difficult and inconvenient techniques are simply abandoned [WW93]. Easy-to-use notations also require less training and cost less to perform system analyses. Thus, it is crucial to shield requirements specifiers from the underlying mathematical complexity as much as possible [End93]. Already, there are FDTs that place more emphasis on usability, such as SCR (based on a tabular notation), Statecharts, SDL and RSML (based on graphical finite-state machines).

If abrupt transitions to formalized requirements are highly unlikely, it is important that FDTs can be incrementally adapted as a part of the development environment without causing large disruptions to the established process. For example, if changes are made in the specification notation for requirements, then design and testing should remain
intact. The technique should allow to retain standard practices like review meetings, version control, etc. The integration of FDTs has been an active area of research [But97, Kem90, WZ91] with the goal to make the transition a seamless process.

On the other hand, a lightweight approach [DD96, JW96, Jon96] to applying FDTs eases the adaptation process by reducing the initial cost of formalization. Emphasizing on partial system specification and analysis, this approach has been demonstrated to deliver moderate benefits, but with a much reduced cost and in a timely manner [ELC+98, Fea98]. For instance, better cost-effectiveness can be achieved by formalizing only those components that deserve more attention [Pre98], increasing the efficiency of the formalization. This selective approach also allows requirements specifiers to stop at any time and get immediate results.

1.5 Formal Description Techniques Studies

Unfortunately, the claims about the recently improved usability and cost-effectiveness of formal methods are still largely unsubstantiated. Indeed, the lack of evidence has been blamed for the slow adoption of FDTs [Tic98]. Thus, there is a need for a large number of relevant case studies in different application domains to exemplify these claims. Not only do the studies demonstrate the approach to realize those claims, but they also provide practitioners with real examples on which their own pilot studies can be based. While there have been published case studies on the use of FDTs, most of them are about safety-critical projects with assumptions that are invalid in the commercial environment. Many studies of commercial projects are of the experimental nature (e.g., with no follow-up) and focus mostly on the technical aspects of FDTs. We believe that the ideal case studies should not only demonstrate the technical possibility of formalizing commercial systems, but also address the practical issues mentioned in the previous section. We feel that the impact of these studies would be more apparent if they contain these activities:

- employ an evolutionary and partial description approach to decrease the cost of the formalization process;
- demonstrate productivity improvements (e.g., increased software quality with the same amount of time);
- apply FDTs to real commercial systems that are being developed and maintained over time to address issues such as scalability and maintainability;
• choose an easy-to-use formal notation that is suitable for the target application domain;
• leverage the investments made in the formalization process to other stages of the software lifecycle to obtain immediate and visible benefits.

1.6 The Origin of the Project

The changing landscape of the telecommunications industry has been putting intense pressure on companies such as Nortel to shorten the development cycles of their products. When a Nortel manager of a software testing team started looking for ways to streamline the software verification process, he found that the quality of the software requirements documents (SRDs) had been affecting the productivity of his team members significantly. Specifically, problems in the SRDs (which he often described as “incomplete, imprecise, inconcise, and ambiguous”) represented a major difficulty in ensuring the quality of the test cases and shortening the test case execution phase. To rectify the situation, the testing team initiated an effort to

“find a way to better capture changing requirements within the current development timeframe, improving the quality of the SRDs and thus aiding system development”.

The team asked us at the Formal Methods Laboratory at the University of Toronto to help investigate the use of FDTs as a possible solution. We were later joined by a Nortel design team who also understood the impact that sub-optimal SRDs have on the design process. Together, we conducted a study to evaluate the feasibility of formalizing requirements in a commercial setting. This thesis describes the study.

1.7 Project Overview

We designed the project to address some of the principles discussed in Section 1.5. The various parts of the project are shown in Figure 1.1. The project placed more emphasis on the economical benefits of FDTs rather and less on their technical feasibility, as virtually

---

5Nortel, for the remaining of this thesis, refers to the Toronto Multimedia Applications Center of Nortel Networks.
any system can be formalized given enough patience and financial resources. Specifically, the goals of the project were on

- finding the means of using FDTs to improve productivity in various stages of the software lifecycle, with the emphasis on deriving test cases from the formal model as the Nortel engineers have expressed concerns about the feasibility of generating code for their proprietary platform.
- exemplifying the costs/benefits of applying FDTs to commercial systems over the traditional development techniques,
- exploring the technical and organizational issues in applying the FDTs to commercial systems and integrating them with the existing development processes, and
- investigating whether FDTs can be economically applied to commercial systems for productivity improvements.

We did not propose any hypothesis for the project because of its pilot nature.

It is crucial to explore the problems in detail before proposing any solution. In the first phase of the project, we conducted an exploratory study with the Nortel engineers to identify the major problems in the Nortel development process, and investigate the causes of these problems and their implications on productivity, as described in Chapter 2. The results from the study allowed us to pinpoint the major areas for improvements and determine if FDTs represented a viable solution.

In the second part of the project, we conducted a formal specification pilot study to exemplify the feasibility of FDTs in a commercial setting. Together with the Nortel engineers, we chose a software system that was representative of typical projects in Nortel and had a good potential to benefit from the study. This process is described in Chapter 3. Using the results from the exploratory study and feedback from the engineers, we conducted a survey to select an FDT that tackled the identified problems most effectively and best met the Nortel engineer's expectations of a usable specification technique. Details of the selection process are presented in Chapter 4. Applying the chosen FDT in a lightweight manner, we specified crucial components of the selected telecommunications system and derived test cases from the resulting formal model, shadowing the existing development process and keeping track of a variety of productivity data for comparison. In Chapter 5, we highlight the important aspects of the formalization and the test case derivation process, the collected data, the experience that we gained, and most importantly, the results, which include a discussion of the changes in software quality and
cost-effectiveness.

After completing the study, we conducted a usability workshop with a group of Nortel engineers to investigate the usability and effectiveness of the FDT used in the study further. Since the workshop was still a work-in-progress at the time of writing, we provide only a brief overview in Chapter 6. Finally, Chapter 7 summarizes this thesis with a discussion of the findings and possible future work.
Chapter 2

A Study of a Commercial Software Development Process

2.1 Overview

The challenges put forward by the manager of a Nortel testing team resemble a common software development problem: specification errors affect different parts of the development process, lowering productivity and eventually the quality of the products. Instead of advocating any particular software engineering techniques immediately to remedy the problem, we felt that it was crucial to investigate the relationship between the quality of the SRDs and the productivity of the development process first.

An exploratory study was conducted at Nortel to research the major problems in the development process and to find out if specification errors were really the root of these problems. Also, by obtaining information such the resource constraints of the development process and how the goals of the development team are prioritized, we could find solutions that tackled directly on the problems more easily. Results from this exploratory study were used to form the basis for the requirements for the formal specification pilot study, described in Chapters 3 to 5.

The rest of the chapter is organized as follows: Section 2.2 provides the necessary background to understand the exploratory study by presenting a brief overview of the Nortel development environment. Section 2.3 describes the organization and the method of the exploratory study. The results from the interviews and their applications to our project are presented in Section 2.4 and Section 2.5, respectively. Section 2.6 provides a summary of the exploratory study.
2.2 Nortel Software Development Process

If we take a simplified view, the software development process at Nortel is a variant of the Boehm spiral model [Boe88] with an acceptance testing phase at the end of the process. Figure 2.1 shows some of the phases that are relevant to our study. At the beginning of the process, the product management team gathers high-level requirements (e.g., "The system should support call waiting.") from customers, competing products, SRDs of existing Nortel systems, as well as the software design team, and produces a document called a commercial specification. Each large system usually has multiple commercial specifications, and the scope of each specification is limited to a particular feature, e.g. the call processing functionality of a private branch exchange.

The design team responds to the commercial specification by constructing a software requirements document that details more concrete requirements from the user perspective, e.g. when an incoming call is detected, the caller should receive a dial tone within 0.5 seconds. The SRD is written mostly in English prose and is occasionally augmented with flowcharts for explaining more complicated requirements. Some preliminary design documents (e.g., software architecture documents) may also be constructed while the SRD is being created. After the document reaches a certain level of completion, it goes through a review process in which designers and sometimes testers inspect the SRD prior to a review meeting and raise their concerns during the meeting. The design team starts designing and implementing the actual system after all reviewers approve the SRD. The developments of the SRD, the design, and the implementation repeat for a few cycles (See the cycle between the phases “SRD creation”, “SRD review”, and “design, implementation, and white-box testing” in Figure 2.1) during which more functionality is gradually specified, designed, implemented, and tested.

When the designers feel that the SRD is reasonably complete (which usually happens one or two cycles before the completion of the final implementation), they finalize and publish the SRD to a centralized document repository. Therefore, while the design team is putting final touches to the design and implementation, the SRD is accessed simultaneously by external groups such as the testing team, the marketing team, and the training team. The testing team, for instance, uses this document to create test cases (see the edge between phases “SRD review” and “test case creation” in Figure 2.1), which are used for verifying the final implementation when the designers feel that the system is ready to undergo acceptance testing. (See the edge between phases “design,
implementation, and white-box testing" and "acceptance testing" in Figure 2.1).

2.3 Method

A multi-case study [Yin94], in which results collected from multiple subjects are compared to obtain the conclusions, was conducted with Nortel software designers and software testers over a period of two weeks. The goals of this exploratory study were to

- identify major problems in Nortel’s development process,
- determine if any of these problems originated from errors in the SRDs,
- investigate whether these problems affected the development process significantly, and
- identify the causes of these problems and their impacts on the development process.

Our exploration study began at the “SRD creation” phase (see Figure 2.1) to avoid over-complicating it with activities such as customer elicitation. Human-related issues, such as organizational behavior, were also ignored.

The study consisted of two main parts: the interview process and the data analysis. Interviews were conducted with the Nortel engineers to obtain their opinions about problems in the development process. After examining their responses using a qualitative data analysis technique, various patterns were identified and their implications to the development process were studied.
2.3.1 Defining the Population

A group of Nortel engineers was asked to volunteer their time to participate in our exploratory study. From those who agreed to participate, three software designers and three software testers that closely reflected the demographics of the actual development process were selected by the managers of the corresponding teams. A summary of their backgrounds is shown in Table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest Level of Education Achieved</td>
<td>Bachelor in Electrical Engineering</td>
<td>Bachelor in Electrical Engineering</td>
<td>Bachelor in System Design Engineering</td>
<td>Bachelor in Electrical Engineering</td>
<td>Bachelor in Computer Science</td>
<td>Master in Information Systems</td>
</tr>
<tr>
<td>Years of Industrial Experience</td>
<td>7 years</td>
<td>3.5 years</td>
<td>12 years</td>
<td>8 years</td>
<td>1.5 years</td>
<td>3 years</td>
</tr>
<tr>
<td>Title</td>
<td>Senior software engineer</td>
<td>Software engineer</td>
<td>Software development team leader</td>
<td>Software tester</td>
<td>Software tester</td>
<td>Software tester</td>
</tr>
<tr>
<td>Previous exposure to formal modeling</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Limited, from university courses</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 2.1: Characteristics of the interviewees.

2.3.2 Formulating the Interview Questions

Even with some defined goals in mind, there were difficulties in formulating the interview questions that were exploratory in nature. As we had no prior knowledge of the Nortel development team or their development process, we could only construct questions that were broad and without a specific focus, reducing the effectiveness of the interviews and the chances of obtaining relevant responses. Ideally, we should have divided the entire interview process into two phases [MH94], in which results from the interviews conducted in the first phase would be used for fine-tuning the questions, and only responses from the interviews conducted in the second phase would be used for the analysis. However, the availability of the interviewees and their tight time constraints eliminated this possibility.

To counter this problem, prior to the interview process, we made a one-week field visit to obtain more information about the software development process by

- observing the activities of the engineers,
• engaging in informal conversations with the engineers,
• browsing through documents that were by-products of the development process, such as the commercial specifications and the test cases, and
• examining the archival records, such as the database that kept track of all implementation errors.

With the additional information, we constructed two sets of interview questions, QS1 and QS2, that were tailored for probing specific problems faced by the software designers and testers, respectively. The first half of both question sets contained similar questions that asked about the educational backgrounds of the interviewees, their experience in modeling requirements formally, and their opinions about the quality of SRDs that they were familiar with. The second half contained questions that were specific to their development environment, shown in Table 2.2. Both question sets contained open-ended questions that allowed the interviewees to elaborate their opinions freely as well as close-ended questions that restricted their responses to a numerical ordinal scale. The actual question sets can be found in the Appendix A.

<table>
<thead>
<tr>
<th>Topics of questions for software designers (QS1)</th>
<th>Topics of questions for software testers (QS2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>problems in the SRDs</td>
<td>implications of problems in the SRDs to the testing process</td>
</tr>
<tr>
<td>causes of problems in the SRDs</td>
<td>problems in the test case creation and execution</td>
</tr>
<tr>
<td>implications of problems in the SRDs to the design and implementation phases</td>
<td>causes of problems in the SRDs</td>
</tr>
<tr>
<td>problems with the specification review process</td>
<td>problems with the test case review process</td>
</tr>
</tbody>
</table>

Table 2.2: Topics of questions in the second half of question sets QS1 and QS2

### 2.3.3 Conducting the Interviews

Question sets QS1 and QS2 were used to interview the software designers (S1, S2, and S3) and the software testers (S4, S5, and S6), respectively, in six separate 45-minute interview sessions. The interview format was similar to the structured interviews described in [MFK90, Wri79] in which the question sets were used as the scripts for guiding the interviews.

All interview sessions were tape recorded. Depending on the actual situations, the interview scripts were not followed exactly. More questions were asked to probe areas that
were of interest, and inappropriate questions were skipped. In addition, the interviewees were encouraged to elaborate on their own opinions about the subjects under discussion. As a result, the scope of their responses was not limited to the subject areas covered by questions in the scripts.

2.3.4 Qualitative Data Analysis

Data collected from the interviews were analyzed using the multi-case analysis method described in [Eis89, MH84]. In our study, each interviewee represented a single unit of analysis (i.e., a 'case'), and their responses were analyzed and compared to reach conclusions.

Relevant high-level constructs, some of which shown in Table 2.3, were identified by comparing the similarities and differences among the responses from the interviewees. For interview questions that were targeted towards a specific team, the scope of the comparison was restricted to interviewees within the same team. Some of the high-level constructs were further broken down for a deeper analysis. For instance, the importance of high-quality SRDs was assessed according to its importance in each development phase, instead of the entire process.

| role of the interviewee in the development process |
| amount of effort spent in different software development phases |
| attitude towards the importance of high-quality SRDs |
| opinions about the quality of the SRDs |
| concerns about the effectiveness of the specification review process |
| problems encountered in creating the SRDs |
| impact of specification errors in the SRDs |
| problems encountered in the creating test cases |

Table 2.3: Some high-level constructs used in analyzing the interview responses.

Then, a data matrix [MH84] with a list of subjects as the row header and a list of constructs as the column header was created to hold the responses obtained from the interviews, which usually contained a mix of qualitative and quantitative information (see Table 2.4 for a mock-up example). For each interviewee, the use of these constructs, rather than the interview questions, as a dimension of the matrix allowed us to group responses that shared similar ideas, discarding duplicate and irrelevant data in this process. Finally, responses from different interviewees with common themes were identified from the matrix using a pattern-matching technique and are described in Section 2.4.
Table 2.4: A sample data matrix.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Construc</th>
<th>opinions about the performance of the Internet browser</th>
<th>opinions about the stability of the Internet browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3</td>
<td></td>
<td>acceptable in most scenarios; considerable slowdown when there are more than five animated GIFs on the screen</td>
<td>unstable; I do not think that the browser should be released now</td>
</tr>
<tr>
<td>S4</td>
<td></td>
<td>slow</td>
<td>unstable; more than 10 errors found in displaying JPEGs; the browser still needs about two months of work, it should be not released now</td>
</tr>
</tbody>
</table>

2.3.5 Threats to Validity

Although effort had been spend to ensure the validity of the study, there were inevitable threats to the validity of the interview, data collection, and data analysis processes. For instance, Yin [Yin94] has identified poorly constructed questions as a major threat to the validity of any interview. To ensure the relevancy of our interview questions to the goals of the exploratory study and to the development process under investigation, we construct our questions using information obtained from the on-site visit, as mentioned in Section 2.3.2. Some questions were intentionally left open-ended to allow interviewees to express their opinions, filling gaps that were missed by the predefined questions. We also tried to preserve the neutrality of the questions by putting the equal amount of emphasis in all areas of exploration.

The confidence in the validity of the data collection process was increased by collecting data from multiple sources and minimizing the effects of possible biased responses from individual subject. The validity was not affected by the small population size, as we never intended to achieve any statistical generalization from the obtained results. Besides, we did not have enough information about the demographics of a typical software development team [GW89] to obtain a proper sampling. Rather, the focus was on identifying problems in a particular development process at Nortel. Also, since all of the subjects were selected by their respective managers, we relied on the managers’ ability in choosing subjects whose opinions reflected the actual condition of the development process and in avoiding subjects that had extreme opinions or a vested interest in the success or failure of the study.

Since the analysis of the responses from the interviews was qualitative in nature,
established analysis techniques were used to prevent our opinions from influencing the results as much as possible. This also made the process of linking data to the results more reliable.

2.4 Results

The six patterns identified from the data matrix are presented in the next subsections using the following format: the Pattern, the Evidence, and the Implications. This format is similar to the presentation of software maintainers’ patterns in [Sim98]. The first part provides a one-line description of the pattern. The second part captures the relevant information from the data matrix as supporting arguments, which include a mix of direct quotes from the interviewees and a summary of their responses. Responses to some of the quantitative questions are presented in tabular form. Unless otherwise stated, the tabular format uses a scale of 1 to 5, where 1 represents “strongly disagree”, and 5 represents “strongly agree”. The last part explains the indirect effects of the patterns on the development environment. Some of the patterns that share similar evidence and implications are grouped together.

In general, the six patterns illustrate that specification errors in the SRDs have a profound and negative effect on the productivity of the development process and the quality of the software systems. On the other hand, it is encouraging that all interviewees acknowledged the importance of high-quality specifications (see Table 2.5) in different development activities.

<table>
<thead>
<tr>
<th>Development phase</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Implementation</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>White-box testing &amp; debugging</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Developing a new revision</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Test case creation</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Test case execution</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2.5: Opinions about the usefulness of a high-quality software requirements specification in the various development phases.

2.4.1 Software Requirements Documents

Pattern 1: Incompleteness and vagueness were two major problems in the SRDs.
**Pattern 2:** The lack of time for a detailed requirement analysis affected the quality of the SRDs.

**Evidence**
When the interviewees were asked to rank the severity of problems, the incompleteness and the vagueness of the SRDs were their two major concerns (see Table 2.6).

<table>
<thead>
<tr>
<th>Problems in understanding the SRDs</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>Avg. (designers)</th>
<th>Avg. (testers)</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verbosity</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>2.67</td>
<td>3.00</td>
<td>2.83</td>
</tr>
<tr>
<td>Wrong assumptions about Readers’ level of understanding</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2.67</td>
<td>3.00</td>
<td>2.83</td>
</tr>
<tr>
<td>Incompleteness</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>3.67</td>
<td>4.33</td>
<td>4.00</td>
</tr>
<tr>
<td>Vagueness</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3.67</td>
<td>4.00</td>
<td>3.83</td>
</tr>
<tr>
<td>Ambiguity</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2.30</td>
<td>2.30</td>
<td>2.33</td>
</tr>
<tr>
<td>Inconsistency</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2.67</td>
<td>2.33</td>
<td>2.50</td>
</tr>
<tr>
<td>Difficult to obtain a high-level overview</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2.33</td>
<td>2.67</td>
<td>2.50</td>
</tr>
<tr>
<td>Difficult to create designs/test cases directly from the SRD</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2.67</td>
<td>3.00</td>
<td>2.83</td>
</tr>
</tbody>
</table>

Table 2.6: Opinions about the major problems in understanding the SRD.

During the interview, an SRD was informally defined as incomplete if some of the requirements were missing or not described in full detail. All designers acknowledged that when defining the functionality of a system during the requirements phase, they usually did not analyze the requirements deep enough to ensure a complete description of the system. S2 recalled that when describing the behavioral requirements of a telephony feature (e.g., requirements that dictate how a phone switch should handle call waiting when a 3-way calling conference is in progress), it was often that only the typical and a few exceptional scenarios were described. Incomplete SRDs are especially error-prone because inexperienced designers or testers may not notice the missing requirements if they only skim through the documents.

The problem is more evident if the SRD describe a subsystem that interfaces with other components in the underlying system. S1 and S5 both described similar situations where the SRD of such subsystem and those of the external components that the subsystem interacted with were being written as the same time, but by different design teams. Since all teams focused on the core functionality of their own subsystems, the external interfaces and interactions were poorly and incompletely documented. Lacking any accu-
rate information, the designers had to specify their subsystems by relying on the limited available information and making unstated assumptions. In addition, they often needed to limit the scope of their SRDs to avoid specifying incorrect requirements as much as possible and inheriting the problem of incompleteness in their own SRDs. Not only did the missing information eventually lead to unhandled scenarios in the implementations, it also caused problems for the testers (see Pattern 4).

To investigate the reasons behind problems in the SRDs (not limited to incompleteness), we compiled a list of rival explanations [Yin94], shown in Table 2.7, for S1, S2, and S3 to identify some possible causes of the problems. Unsurprisingly, the lack of time was the dominant reason, given the company's recent change of focus to rapid software development. While S1, S2, and S3 felt that the time constraint applied to the entire development process and not just the requirements phase, the pressure to produce tangible results (e.g., prototype or working code) often left them with little time for analyzing the requirements. In fact, S1 believed that "although it is very important to understand our requirements and perform preliminary requirements analysis before attempting to construct any design, it is an ideal theory that has yet to be put into practice." The designers were aware of the problem of incompleteness, but they often did not have the time to search for the missing requirements nor to analyze them. S3 admitted that if the designers "have the luxury of time to have the majority of the requirements specified correctly and to have them analyzed deep enough, many of the design and implementation changes could actually be avoided, possibly shortening the length of the overall development process".

<table>
<thead>
<tr>
<th>Causes of problems in the SRDs</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engineers have no incentive to create high-quality SRDs</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1.67</td>
</tr>
<tr>
<td>The current level of quality is perceived to be adequate</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2.67</td>
</tr>
<tr>
<td>Insufficient time</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4.00</td>
</tr>
<tr>
<td>Complexity of the requirements</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3.00</td>
</tr>
<tr>
<td>A lack of guidelines for specifying the requirements</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2.33</td>
</tr>
</tbody>
</table>

Table 2.7: Some possible causes of the problems in the SRDs.

Consequently, S1 mentioned that designers were forced to construct vague requirements, in order to record as much relevant information as possible in the shortest amount of time. For instance, when describing the interactions between a voice mail system and its users, only a few pages of prose were given to provide a general idea; information such
as exception handling, the lengths of various types of idle timeouts, and the length of pauses between the voice prompts would only be mentioned in inadequate detail. The rationale is that there should be more time or knowledge to clarify those vague requirements in future. However, in reality, this usually does not happen until late in the design phase, possibly after many avoidable requirements and design changes.

Although vague requirements are easier to detect than incomplete requirements, their effect on the development process is equally problematic. For instance, S2 found that vague requirements had caused a lot of confusions among readers who might not be involved in the requirements phase (such as himself); they needed to spend extra time to contact the original author for clarifications. Thus, it was difficult for him to ensure that the vague requirements were completely and correctly implemented.

**Implications**

Inadequate emphasis on the requirement analysis leaves avoidable specification errors that affect the quality of the corresponding design and lengthen the entire development process. The scope and the severity of these problems are magnified by the fact that external groups also access the same SRDs. Therefore, a tremendous reliance is placed on the specification review to ensure early identifications of these problems and prevent them from propagating to other development phases.

### 2.4.2 Manual Inspections

**Pattern 3:** Faults in the manual inspection process affected its ability in detecting specification errors.

**Evidence**

In the development process that we investigated, manual inspection of the SRDs is the only means of ensuring correctness of the requirements and detecting errors overlooked by the requirements specifiers. However, results from the interviews indicated that faults in the inspection process have reduced its effectiveness in achieving the intended objectives. Specifically, the major problems lie in the lack of guidelines for the inspection process and the fact that many reviewers do not treat the process seriously (see Table 2.8).

Prior to the review meeting, reviewers are often asked to inspect a large document (e.g., more than 200 pages) in isolation with no specific instructions. S1 described the
Table 2.8: Opinions about problems in the manual inspection process.

<table>
<thead>
<tr>
<th>Problems in the manual inspection process</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitations of the manual inspection</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3.67</td>
</tr>
<tr>
<td>A lack of guidelines for directing the inspection</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4.33</td>
</tr>
<tr>
<td>Insufficient time</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3.67</td>
</tr>
<tr>
<td>Reviewers treat the inspection process like a formality</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.00</td>
</tr>
</tbody>
</table>

major difficulty with the inspection was the lack of specific instructions; reviewers often “did not know where to start and what they were expected to do”. They performed the inspection at their own discretion, which ranged from “start from the beginning and read until the end” to “review only the important sections”, thereby strongly correlating the quality of the review with the patience and the experience of the reviewers. The lack of instructions was particularly problematic for the less experienced reviewers.

Other problems in the SRD, such as vague requirements, also reduce the effectiveness and the level of detail of the inspection, prompting the reviewers to treat the inspection process as a formality. For instance, in one of the SRDs that we surveyed, some of the requirements were so vague that it was difficult to infer any sense of correctness from them. S4 even questioned the value of performing detailed reviews on SRDs that were full of problems such as those mentioned in Pattern 1. Because of these concerns, S3 acknowledged that many reviewers had slowly lost faith in and devoted less attention to the inspection process. While the company policy ensures the existence of the review meetings, S3 recalled that many reviewers only skimmed through the SRDs. The inspection process diverts from the original purpose of error detection to a general cross-checking for ensuring that the SRD had the right concept—a common phenomenon that is also described by Jones in [Sai96]. Performing this type of high-level checking at such a late stage of the requirements phase is inefficient, as problems identified tend to have a larger impact and are more costly to rectify.

**Implications**

Being the only method of error detection, ineffective reviews affect the development process significantly. Severe but subtle errors can easily slip into the 'finalized' versions of the SRDs, propagating to the design phase and affecting external groups that access the document. For instance, during the interviews, members of the testing team indicated
that their test cases were inevitably affected by the quality problems of the SRD (see Pattern 4). The persistence of these specification errors also creates a continuous need to update the SRDs after the requirements phase (see Patterns 5 and 6).

2.4.3 Testing

**Pattern 4:** The quality of the SRDs and the time constraints in the test case creation phase were two major obstacles to an efficient testing phase.

**Evidence**

Depending on the SRDs for a part of their work, testers are clearly concerned about their quality. We compiled a list of rival explanations for S4, S5, and S6 to identify possible problems that they encountered when creating test cases. Table 2.9 shows their responses, which indicate a strong correlation between the quality of the test cases and that of the SRDs. Specifically, the incompleteness and the vagueness of the SRD are the major concerns (see Table 2.6 in Pattern 1).

<table>
<thead>
<tr>
<th>Problems in creating high-quality test cases</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testers were unwilling to create detailed test cases</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1.67</td>
</tr>
<tr>
<td>Problems in the SRD</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4.33</td>
</tr>
<tr>
<td>Inadequate time</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3.67</td>
</tr>
<tr>
<td>A lack of guidelines to direct the test case creation process</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3.00</td>
</tr>
</tbody>
</table>

Table 2.9: Opinions about problems in creating high-quality test cases.

Missing requirements in the SRDs often translate to missing test cases, which S5 simply described as “no requirements, no test cases”. Unless testers are experienced and have the time and the patience to analyze the requirements extra carefully before creating test cases, the coverage is affected. Even under this scenario, S4, an experienced tester, recalled a time where he tried to consult the design team for a crucial but missing requirement. Much to his surprise, the design team knew about the absence of the requirement; however, they had not made a decision and asked him for opinions instead. In general, if testers follow the SRDs naively, there is little guarantee that the system will be tested properly or completely.

Vague requirements also make the SRD more difficult to understand, lengthening the test case creation process. S6 recalled an extreme scenario where the requirements were
so poorly documented that she had to spend extra time to read the design document to prepare herself for the test case creation phase. Typically, SRDs do not contain enough information for creating detailed test cases. Testers need to make unstated assumptions in order to proceed, and incorrect test cases may be created in this process. Alternatively, vague test cases (e.g., test cases without clear and detailed descriptions of the test steps and the expected system responses) are often used as a temporary solution, inheriting the problems from the SRD. Most testers do not feel confident about the correctness of their test cases, as the quality can only be good as the SRDs.

Time constraints, unlike problems in the SRDs, have a smaller impact on the quality of the test cases. As shown in Table 2.10, they affect only the level of detail of the test cases but not their test coverage much. S6 explained that in general, testers are simply not allowed to leave things out intentionally. If time is tight, they may not test all combinations of the functionality, but they always try to achieve full (functionality) coverage.

<table>
<thead>
<tr>
<th>Tester-specific statements</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fewer test cases are created when time is tight.</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1.67</td>
</tr>
<tr>
<td>Vague test cases are created when time is tight.</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3.67</td>
</tr>
</tbody>
</table>

Table 2.10: Responses to tester-specific statements.

Ideally, testers attempt to specify as much details in the test cases as possible. However, S4 and S5 explained that the test plan schedule and the staff availability were usually much tighter in the test case creation phase than the execution phase. In order to meet the deadlines, they were forced to “create more general test cases and let the executor figure the rest out”.

**Implications**

Problems in the SRDs affect testers in both creation and execution of test cases. While not all quality problems of the test cases can be traced back to specification errors, testers express frustrations about their dependency on the design team and their inability to rectify the situation. The work of the testing team depends largely on the completeness and accuracy of the SRDs. They often argue that if designers spend more time analyzing the requirements, their productivity can be improved both in terms of the quality of the test cases and in the length of the testing process.
However, from the perspective of the designers, high-quality SRDs are desirable but not critical. Having the resources and background knowledge to resolve the specification errors independently and quickly, the design team is less affected by the quality of the SRDs. Thus, there are fewer reasons for them to spend the extra time to construct higher-quality SRDs up front. All interviewees agreed that the definition of "acceptable quality" is an area of disagreement between the design and the testing teams.

2.4.4 Propagation of Specification Updates

**Pattern 5:** The lack of frequent updates to the SRDs affected the testing phase and the maintenance of the SRDs.

**Pattern 6:** Changes in the SRDs were sometimes not reflected in the corresponding test cases.

**Evidence**

Figure 2.2 illustrates the relationship between various development artifacts, as described by the interviewees. As Nortel designers make design or implementation changes, there is a continuous need to propagate the resulting modifications to the SRDs. However, at this stage, most designers have already diverted their focus from the SRDs onto the actual designs. For instance, designers, such as S3, only treat the SRDs as a starting point for the design phase and do not use them after gaining the required knowledge. He admitted that in many instances, it was more time efficient for him to rectify the errors discovered by groups such as testing rather than to find the errors actively. Besides, as suggested by the number of changes that S1 observed in his project, designers update the SRDs only for major changes and only if their schedules allow.

The lack of frequent updates is evident from how testers are affected. S4, S5, and S6 complained that it was very difficult for them to create test cases for the final implementations without any modification. Even after re-reading the latest versions of the SRDs (which were usually not that recent) at the beginning of the test case execution phase to re-check the coverage and the validity of the test cases, they still encountered many discrepancies between the SRDs and the implementations when the test cases were actually executed. S5 explained that she often needed to use her judgment to decide whether the implementations were indeed faulty. The ideal situation was to consult the designers for clarifications, but their tight schedules as well as hers prevented them from meeting
frequently. S6 recalled an extreme case where half of the test cases that she created were invalid, making detailed consultations with the designers an impossible alternative.

In those occasions where the designers update the SRDs, the changes need to be reflected in all documents that were created based on the outdated SRDs. While there are established processes to notify the affected groups, they must be willing to spend extra time in rectifying problems caused by others. For instance, S4, S5, and S6 complained that their test cases became obsolete too fast, and too much work was required to re-read the SRDs and update their test cases. Unfortunately, the test cases were usually not updated until the end of the entire test cycle. Depending on the size of the project, the cycle can be four to six months long, increasing the risk that some test cases may never be updated accordingly and reducing the confidence in using the test cases as the authoritative reference for determining the correctness of the implementations.

The maintenance of the SRDs is also affected. At the end of the development cycle, designers allocate a few days to go through the SRDs and resolve all outstanding issues before moving onto the next release. However, according to S1 and S5, there is no guarantee that all errors, especially the subtle ones, are rectified. S1 described a large system that went through 13 releases. Between each release, the designers created a new SRD by migrating content from an earlier version that might not accurately reflect the requirements of the implementation; unsolved issues propagated to the new SRD. S1 joked that in order to obtain a complete understanding of the entire system, a person must have the patience to read all 13 SRDs and to combine relevant pieces from them.

**Implications**

As illustrated in Figure 2.3, many of the development problems identified in our study
originates from the lack of time, which has a direct effect on the quality of the SRDs and the test cases (see Patterns 1 and 4, respectively). After the requirements phase, the general unwillingness of the design and the testing teams to rectify problems in these artifacts also enhances the impact of the specification errors. Clearly, the productivity of the testing team affected. In cases where the implementations do not correspond to the requirements, it is likely that the test cases can never detect some of their errors.

As experienced members leave the design team and the system grows in size and becomes more complicated over time, the gap between the SRDs, the test cases, and the implementations widens, becoming increasingly difficult to synchronize them. Slowly, more parts of the entire development process are affected in this vicious cycle.

![A vicious cycle diagram](image)

**Figure 2.3: A vicious cycle.**

### 2.5 Applying the Results

The six patterns and the discussion of their evidence and impacts allowed us to address the goals of the study described in Section 2.3. The major finding was that due to time constraints, specification errors affected the productivity of the development teams and the quality of the software systems significantly. Another finding confirmed the importance of high-quality SRDs to the development process and assured us that specification errors were indeed a real and recurring problem. Since most of the identified problems revolved around the SRDs, we felt that conducting a pilot study to investigate the use of alternative requirements specification techniques was well-justified. However, these techniques should be used without compromising the length of the development process.
This pilot study should demonstrate that the new specification technique provides significant improvement over the existing method. To tackle the identified problems directly, the patterns were used to form the main requirements of the pilot study. Besides fulfilling the high-level goals mentioned in Section 1.7, the study should also tackle the issues below:

- **Quality improvement of the SRDs.**
  The specification technique should help the designers lay out detailed requirements in an easy to understand manner and encourage them to perform thorough analyses. By eliminating many of the problems identified in Pattern 1 early in the requirements phase, the quality of the SRDs should be improved, benefiting external groups tremendously.

- **Improvement of the inspection process.**
  Being the only method of error detection in the requirements phase, the inspection process is certainly crucial in preventing errors from propagating to other development phases. While its effectiveness can be enhanced by having precise and correctly written requirements, a specification technique that structures the SRDs in an easily-reviewable manner can better guide the reviewers to perform detailed cross-checking.

- **A stronger link between the SRDs and the test cases, and possibly between the SRDs and its corresponding designs and implementations.**
  The impact of the specification errors was evident when the SRDs were not actively updated or when their changes were not reflected in other development artifacts. With a tighter link between them, it is easier to ensure that the requirements are applied properly; updates to the SRDs can also be propagated more easily. These should increase the value of the SRDs in later development phases and encourage the designers to maintain them more effectively.

- **Testing Support.**
  Given the close relationship between SRDs and test cases, specification techniques that have explicit testing support can benefit the testing team directly. For instance, they should encourage designers to specify detailed requirements from which testers can easily create accurate test cases with better coverage. Also, the use of the same specification in the testing phase amortizes the investments made in the requirements phase.
- More effective use of time.

As results from the interview study showed, time was the most scarce resource; thus, it is imperative that the new specification technique does not lengthen the development process, even if the quality of the SRDs is improved. The focus should be on achieving the best possible quality within the current development timeframe.

These requirements had a great influence on how the pilot study was conducted, such as the criteria used in choosing a suitable specification technique, the approach taken to apply this technique to specifying a Nortel software system, and how the results from the pilot study were interpreted. Details of this study are described in Chapters 3 to 5.

### 2.6 Summary

In response to the challenges put forward by the manager of a Nortel testing team, a multi-case study was conducted to examine the Nortel development process. We interviewed six engineers to identify the major development problems, analyze their causes, and investigate their impact on the productivity and on the quality of the resulting products. By using a qualitative data analysis technique, we identified six patterns of the development process. These patterns allowed us to conclude that errors in the SRDs indeed constitute a major part of the problems faced by the Nortel manager. Another finding confirmed the importance of the SRDs to the development process.

We believe that these results provided enough justification to conduct a pilot study, designed to investigate the use of a better requirements specification technique for solving the identified problems. The six patterns were used to form the basis for the requirements of the pilot study.
Chapter 3

Software System Selection

3.1 Overview

Results from the interview study in Chapter 2 indicate that specification errors have a profound impact on the productivity of the development team and the quality of the resulting products. A pilot study was conducted to investigate the possibility of applying an alternative requirements specification technique to rectify the identified development problems. This chapter describes the first phase of the study in which an appropriate Nortel software system was chosen. We explain the selection criteria in Section 3.2. Section 3.3 provides an overview of the chosen system as well as descriptions of two system components that are analyzed in this thesis.

3.2 System Selection

Choosing the appropriate system constitutes an important part of the study. As not all systems are worth engineering [Pre98], only some systems are worth formalizing. For instance, there can only be limited benefits in formalizing a GUI system when simpler methods such as prototyping are better at handling systems with frequently changing requirements. Thus, it was crucial to select a system that had a good potential to benefit from the formalization process; this was particularly important because of the exploratory nature of our study. We felt that the impact of our study would be more apparent if the system to be modeled

- had relatively stable requirements;
was not too broad in scope but contained complex parts;

- was relatively self-contained and had loose coupling with the underlying system;
- was reactive—almost all inputs resulted in outputs that accurately indicated the state of the system [SS98].

While the criteria certainly do not apply to all commercial systems, they increase the chance of success of the study, which is essential for a project of pilot nature. To make the project meaningful, we did not want to be directly involved in choosing a system, hoping to work on something representative of typical projects in Nortel. We provided these criteria as suggested selection guidelines to a group of Nortel engineers, consisting of software designers and software testers, and they decided that we should work on a system called ServiceCreator.

### 3.3 System Description

ServiceCreator is a subsystem in the Operation, Administration and Maintenance (OAM) software of a multimedia-messaging and call-processing system, connected to a private branch exchange (PBX). It is a voice service creation environment where PBX administrators can build custom telephony applications, such as an automated technical-support call redirector, without worrying about the underlying complexity of a phone switch. Due to the non-disclosure agreement (NDA) with Nortel, we provide a mock-up view of the ServiceCreator in Figure 3.1.

The left panel of ServiceCreator shows some of the voice-service components available to administrators; the right panel shows the graphical workspace where administrators can connect the predefined components, such as the voice menu, fax selection, call transfer and announcement, to build telephony applications. For example, a telephony application shown in the right panel lets callers select a password-protected fax document. When the application is activated, a call session begins at the Start component, and a caller is required to enter a numerical password in order to select the fax document associated with the FaxSelect component. She is directed to the End component if the maximum number of incorrect entries is reached. In both scenarios, the call session ends when the End component is reached.

The lines connecting various components represent the potential control flow of the call session, and the actions performed by the caller in an active component determine
the output path taken. For instance, in the component *PasswordCheck*, the caller exits via the path *Password* if a correct password is entered, or the path *Max. Invalid* if there are too many invalid password attempts.

During run-time, the telephony application keeps track of a variety of persistent data about the call session such as the number of invalid password attempts and whether touch-tone keys have been received. Some components make use of this information in determining the appropriate output path. For instance, if a caller stays idle in a voice menu, the timeout routine for rotary callers is activated if no touch-tone keys have been received in the call session. In the other case, a timeout prompt is played to encourage the caller to take action. Thus, although many voice-service components are shown as distinct entities in the graphical workspace, they affect the behavior of each other by accessing and modifying persistent data during the actual execution.

A "medium" level of assurance is required in ServiceCreator: while the software quality must be higher than typical office productivity applications such as word processors or schedulers, it does not need the robustness possessed by safety-critical systems such as railway control systems. The Nortel engineers explained that they did not attempt, nor it was necessary, to attain absolute software quality. Rather, their goal was to ensure that ServiceCreator met the expectations of the customers using the least amount of
resources.

In our study, we analyzed the run-time behavior of 15 ServiceCreator components, described by an 80-page natural language specification (out of a total of 23 components with 97 pages of requirements); a brief description of these components is given in Table 3.1. However, due to the NDA, we focus our discussion on the analysis of PasswordCheck and ThruDial, which are of the medium complexity relative to the other 13 components. In the sections below, we describe the specifications of these two components. Note that the description is quite different from the original Nortel specifications, shown in Appendix B. Our presentation is shorter and clear of known requirements errors. It is provided for information only: the analysis described in the remaining of the thesis is based on the original specifications.

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>denotes the beginning of an application and performs the necessary initializations</td>
</tr>
<tr>
<td>End</td>
<td>terminates the application and performs necessary cleanup</td>
</tr>
<tr>
<td>RotaryDial</td>
<td>directs rotary callers to a predefined attendant for assistance</td>
</tr>
<tr>
<td>Announcement</td>
<td>plays a voice prompt</td>
</tr>
<tr>
<td>SetLanguage</td>
<td>changes the default spoken language of the application</td>
</tr>
<tr>
<td>Menu</td>
<td>lets a caller make a touch-tone selection</td>
</tr>
<tr>
<td>PasswordCheck</td>
<td>controls access to certain parts of the application by predefined passwords</td>
</tr>
<tr>
<td>DateControl</td>
<td>controls access to certain parts of the application by the current date</td>
</tr>
<tr>
<td>DayControl</td>
<td>controls access to certain parts of the application by the current day of the week and predefined holiday schedules</td>
</tr>
<tr>
<td>TimeControl</td>
<td>controls access to certain parts of the application by the current time</td>
</tr>
<tr>
<td>FaxSelect</td>
<td>lets a caller select a fax document</td>
</tr>
<tr>
<td>FaxSend</td>
<td>sends the selected fax documents to a caller</td>
</tr>
<tr>
<td>CallTransfer</td>
<td>transfers a caller to a predefined phone extension</td>
</tr>
<tr>
<td>ThruDial</td>
<td>lets a caller transfer herself to another phone extension</td>
</tr>
<tr>
<td>ImportApp</td>
<td>allows an application to be represented as one voice-service component, so that it can be reused in another application as an embedded component</td>
</tr>
<tr>
<td>Continue</td>
<td>directs the control flow of a terminated embedded application back to its parent application</td>
</tr>
</tbody>
</table>

Table 3.1: The 15 ServiceCreator components analyzed in our study.

3.3.1 PasswordCheck

The PasswordCheck component is described by a 5-page natural language specification. Figure 3.2 shows the workspace view of this component. When a caller first enters
PasswordCheck, an initial prompt is played. The digits entered by the caller are then validated against any of the passwords (up to five) defined by the administrator. For instance, the path Password 1 is taken if the entered digits match the first defined password. The caller is forced to leave the component using the Max. Invalid output if the maximum number of invalid password attempts is reached. Such attempts are also monitored on a per-call session basis, and the caller leaves via the Max. Invalid/Session output if the per-call session limit is reached. The caller can also enter the * key to retrieve the help prompt, which has a side effect of clearing the password entry, or the # key to exit prematurely via the # (cancel) path if no password has been entered. If the caller stays idle for a certain time period and has not previously keyed in any digit prior to entering PasswordCheck, she is assumed to be using a rotary phone and is transferred to the RotaryDial voice-service component (not shown in the Figure). Otherwise, one of the two delay prompts may be played depending on whether she has begun keying in the password. After two more timeouts, the caller exits via the No Response path.

Figure 3.2: The workspace view of PasswordCheck.

When an administrator creates an application in ServiceCreator, she can modify the default behavior of each PasswordCheck component using the graphical dialog shown in Figure 3.3. For instance, the drop-down box titled Password Prompt lets the administrator choose the initial prompt out of a system default prompt, an empty prompt, or a user-defined prompt. The textbox titled Number of invalid response retries controls the maximum number of invalid password attempts before a caller exits via the Max. Invalid output. This value is computed on a per-component basis and is clear when the caller exists the component, while that of the Max. Invalid/Session output is kept track of on a per-call session basis. The latter is a system-wide setting that is not modifiable using the PasswordCheck dialog. Also, the administrator can define up to five numerical passwords for each PasswordCheck component. To set up a password (e.g., Password 1), the administrator checks the box titled Password 1 and fills in the desired numeri-
cal password in both the *Password* and the *Confirmation* columns. A minimum of one password must be defined.

![Property dialog of PasswordCheck](image)

**Figure 3.3:** The property dialog of PasswordCheck.

### 3.3.2 ThruDial

The ThruDial component, described by a 10-page natural language specification, acts like an automated attendant that allows a caller to transfer herself to another directory number (DN) by keying in the names or the DNs of the persons that they wish to call. Similar to PasswordCheck, an administrator can modify the behavior of ThruDial using the graphical dialog shown in Figure 3.4. The checkbox titled *Use fixed length numbers* and the three radio boxes inside the *Dialing options* group limit the caller to one of the input methods to locate a DN: variable-length number dialing, fixed-length number dialing, name dialing, or “number or name”dialing. Details of these dialing modes are explained later in this section. The drop-down box titled *ThruDial greeting* lets the administrator choose the initial greeting prompt out of the system default prompt, an empty prompt, or a user-defined prompt. All ThruDial prompts are interruptible, and the content of the system prompt varies depending on the input method chosen. Using the drop-down box at the bottom of the dialog, the administrator uses an access-control list to define the set of phone numbers that the caller can/cannot access; this list can
only be created in the OA&M, i.e. not within this dialog.

![Figure 3.4: The property dialog of ThruDial.](image)

Figure 3.4: The property dialog of ThruDial.

Figure 3.5 shows the workspace view of ThruDial. The caller leaves via the \# (Cancel) output if she presses the \# key prior to entering any digits. The No Response output is taken if she stays idle for too long; the detailed exiting condition varies depending on the input method chosen. The caller leaves ThruDial via the 0 (Attendant) output if she enters

- “0” as the first entry and then waits for a timeout in number dialing,
- “0#” in number dialing, or
- “0” as the first entry in name dialing.

The output path Success is taken if the caller is successfully transferred to the requested DN. The small rectangle titled End at the end of this path indicates the call session, with respect to the ServiceCreator application, is terminated. This path is also taken if the DN requested by the caller is invalid but has an associated mailbox; ThruDial transfers her to this mailbox. However, if there is no mailbox, or if the DN is blocked by the restriction/permission list, the system plays a prompt and then lets the caller start over and enter another DN. If the DN is valid but the transfer fails because of hardware problems, the caller is given a chance to retry or exit using \# key. In the case where the DN is valid but busy, the system plays an error prompt to indicate this status.
Number Dialing

When the radio box titled *Allow dial by number* is chosen by the administrator, a caller enters the DN that she wishes to call. If the *Use fixed length numbers* box is not checked, she can enter a variable-length DN with up to 15 digits. The caller must terminate the entry using the # key, and ThruDial transfers her to the DN entered. If that box is checked, she is required to enter a fixed-length DN; the administrator specifies this length, which has a range of 1 to 13, in the graphical dialog. After the caller enters enough digits, ThruDial performs the call transfer automatically without waiting for a # key. However, if the caller presses this key prior to fulfilling the fixed-length DN requirement, the system plays an error prompt.

Optionally, the administrator can specify a string of digits, with a maximum length of six, to be padded to the left of the caller's entry. Consider the case where the administrator specifies the fixed length to be five and the padding string to be “123”. If the caller enters “45#”, the system transfers her to the extension “12345”. Also, starting from the left of the padding string, only as many digits are appended to fulfill the fixed-length DN requirement. Using the same example, the resulting entry becomes “19876” if the caller enters “9876#”. However, if the caller presses the # key to terminate the number entry when the length of the padded number is less than the specified fixed length (e.g., the caller only enters “4#”), ThruDial plays the same error prompt described in the last sentence of the previous paragraph.

In any number-dialing state, the caller can retrieve a help prompt by pressing the * key as the first digit entry. Pressing this key in other situations adds a pause to the number entry. Also, in the case where the caller stays idle in the fixed-length dialing mode, ThruDial performs the transfer immediately if the length of the padded number is greater than or equal to the specified fixed length. However, if there are not enough digits or if the caller is in the variable-length dialing mode, ThruDial continues with the normal timeout sequence and plays a prompt to encourage the caller to take action. After two more timeouts, the caller exits via the *No Response* output.
Name Dialing

When the Allow dial by name radio box is chosen, a caller keys in the name of the person that she wishes to call, starting from the last name. The OA&M system keeps a list of user names and their DNs. As soon as ThruDial finds a single match, the system plays a prompt and transfers the caller. Even when an exact match is not found, the caller may terminate the name entry by pressing the # key. If more than nine matches are found, the system plays an error prompt. Otherwise, thru dial announces the number of matches and then plays a prompt to let the caller choose one of the people to call. For example, the prompt for three matches is

“For (person1), press 1; for (person2), press 2; for (person3), press 3.”

If available, ThruDial plays the pre-recorded identifications of (person1), (person2), and (person3). Otherwise, their names are spelled out to the caller. The caller presses the corresponding key to be transferred to one of them.

In any name-dialing state, the caller can obtain a context-sensitive help prompt by pressing the * key. The content of this prompt depends on whether she has begun keying in digits. The timeout sequence mirrors that of the number dialing except that a different set of timeout prompts is played. Also, during each timeout, ThruDial attempts to find a single match, and transfers the caller if the attempt is successful. Otherwise, the normal timeout sequence occurs as usual.

Number or Name Dialing

If the administrator chooses the Allow dial by number or name option, the caller is free to locate the person to call by keying in the DN directly or by spelling her name. In the latter case, the caller needs to enter a special dialing prefix prior to entering a name. This prefix is a system-wide setting and is defined in the OA&M.

3.4 Summary

Together with the Nortel engineers, we chose a software system, called ServiceCreator, which was representative of typical projects at Nortel and had a good potential to benefit from the formalization process in the pilot study. An overview of ServiceCreator and two
of its components is provided to enhance the understanding of the study. The original Nortel specifications of the components can be found in Appendix B.
Chapter 4

Formal Description Method Selection

4.1 Overview

A successful formalization of a system in a commercial setting depends crucially on a suitable formal language, supported by an appropriate tool. Thus, we use the term _method_ to indicate both the language and its tool support in this chapter. In the second phase of the formal specification pilot study, we conducted a formal description method survey to choose a method that could effectively tackle the identified development problems, was usable in the Nortel development environment, and was suitable for modeling telecommunications applications. By performing a careful selection, we hoped that the chosen method could increase the impact and the chance of success of our pilot study while ensuring its relevancy to the Nortel development environment.

After explaining the rationale of using formal description methods as a solution in Section 4.2, we discuss the preliminary screening that limited our search for suitable methods to three candidates in Section 4.3. From Section 4.4 to Section 4.6, we describe the process in which we modeled a part of the ServiceCreator using the three methods and selected a winner among them. Section 4.7 provides a summary of this chapter.

4.2 Rationale for Using Formal Description Methods

If results from the interview study indicate that many development problems revolve around creating, understanding, and maintaining the specifications, it is logical to con-
centrate on improving how requirements are specified and presented. In general, formal description methods are suitable for increasing quality of the requirements and documenting system behavior precisely and concisely, and these properties seem to be suitable for tackling the problems in the Nortel development process. The list below discusses their suitability by referring to the major patterns identified in the interview study.

- **Requirement Specifications**

  Applying formal description methods to requirements have been shown to increase the overall system quality [EC98, Fea98]. The process of formalization helps detect ambiguities and inconsistencies present in the requirements by enforcing a formal syntax. In addition, formal description methods that provide a better sense of completeness, e.g., those that are based on a finite-state machine (FSM) notation, can mitigate the impact of the two major problems identified in Pattern 1 of the interview study. In fact, two designers have already found natural language descriptions to be inadequate in expressing complex behavioral requirements precisely and concisely, and have started using other specification techniques such as flowcharts.

- **Manual Inspections**

  Formal description methods do not replace the need for a specification review, especially when there is no intention to specify all requirements formally and verify their correctness and consistencies. However, the use of these methods increases the quality of the specification, e.g., eliminate vague requirements. This helps reviewers understand the SRD better and encourage them to devote more attentions to the reviews. The formal notation also allows them to argue about the correctness of the requirements more precisely. In general, formal description methods helps make specification reviews more crisp and useful in locating problems as inspections performed late in the software cycle (e.g., code reviews) [Jon96], while avoiding the cost of rectifying late errors.

- **Traceability Support**

  Patterns 5 and 6 of the interview study exemplify the adverse effects of the discrepancies between different software artifacts. While they involve issues (e.g., tight time schedules and different goals of the design and testing teams) that are beyond
the scope of our project, formal description methods that provide tighter links between different development phases help reduce these problems. For example, if requirements are specified using a FSM-based notation at a proper level of abstraction, the model can serve as a basis for the preliminary design, and design decisions can be traced back to the requirements. Moreover, tools that let testers derive test cases from the specification can provide traceability between the requirements and the test cases, benefiting the testing phase both initially and as the requirements change.

From the above discussion, we believe that formal description methods represent a well-suited (but certainly not the only) solution for the identified problems. Nevertheless, these claims can only be substantiated by conducting a case study where a formal description method is applied to a real Nortel software system in a commercial environment. Ultimately, the success or failure of the study determines the validity of these claims.

4.3 Preliminary Screening

During the first phase of evaluation, we conducted a broad screening of formal description methods to survey the available choices. We intentionally loosened our criteria at this stage because of the preliminary nature of the screening. Clearly, easily readable and reviewable artifacts [Par96], a simple notation [Hei98], and strong tool support were some basic requirements for formal description methods to be usable in a commercial setting. Using results from the interview study, we also looked for methods with requirements modeling and testing support as well as traceability support between these two phases. Last, the method should be suitable for describing stimulus-response systems, such as ServiceCreator.

We began our search by assessing reviews and product information on the Internet. For methods that matched our criteria closely, we obtained additional information from the vendors and evaluation copies, if possible; details of the screening process can be found in [Won98]. Surprisingly, these criteria turned out to be extremely limiting as most of the methods surveyed had either just modeling or just testing support, did not have a formal notation, or were simply too difficult to be used in industry. We eventually had to relax our criteria and to consider tools that provided some modeling
and testing support. The scope of our search was eventually narrowed down to three industrial methods: SDT [Tel98] from Telelogic, Validator/Req [Aon98] from Aonix, and TestMaster [Ter98] from Teradyne.

4.4 Method

To perform a more detailed assessment, we structured our evaluation as a multi-project study [BSH86] where the three methods were used to model a simplified version of PasswordCheck using the specification shown in Appendix B.1. The simplifications are as follows:

- the component was assumed to be the only component in the application;
- the component could use only the default initial prompt;
- the behavior related to persistent data was not modeled (e.g., rotary caller behavior);
- the component could have only one password.

Because of numerous omissions and ambiguities in the original specification, consistent assumptions were made throughout the entire modeling process. The three resulting models can be found in Appendix D.

We identified the strengths and weaknesses of the different methods by applying them to the same component. Section 4.6 provides a brief overview of the evaluated methods. Afterwards, we focused on evaluating the suitability of the methods against a more comprehensive list of criteria. Based on the feedback from the Nortel engineers, our impressions of the tools, and the models produced, we applied a quantitative feature analysis technique [KJ97a] to choose the most suitable method, presented in the next section.

4.5 Quantitative Feature Analysis

In feature analysis, a winner is chosen by first evaluating the candidates against a list of relevant criteria. Then, the results are aggregated so that a final decision can be made easily. This type of evaluation helps clarify important features of the methods in the context of the organization and the environment, identify the differences between the methods, and justify the conclusion objectively. Similar evaluations of formal description
methods have been conducted [Ard97, CM95, CGR93, vKKRS98, KDGN97] in different application areas. However, unlike these research studies, our evaluation tied much closer to the actual development environment, and we could not propose a set of 'relevant' criteria without actually addressing their relevance to the development environment and the rationale for using them. Also, since Nortel engineers and managers were expected to review our evaluation and made the final decision, the fully qualitative evaluation and aggregation approaches common in these research studies were not appropriate; they would add unnecessary burden to the Nortel personnel for tracing the rationale and arguments that lead to the conclusion quickly and precisely.

An emphasis of our evaluation was the relevancy of the criteria. Thus, we derived our main evaluation criteria according to the feedback from the Nortel engineers as well as from the problems that we identified in their development environment. The methods were then evaluated against the criteria using numerical scores and qualitative justifications, allowing the Nortel personnel to interpret each detailed evaluation more unambiguously. Weightings were assigned to the criteria to reflect their relative criticality\(^1\). The conclusion of the entire evaluation was reached by comparing the final scores of the methods, obtained by adding weighted scores from each criteria. After discussing the results from the interview study and the above evaluation plan with the Nortel engineers, it was decided that we should conduct the evaluation with the following assumptions:

- the formal description method would be applied to ServiceCreator (or other similar reactive telecommunications systems) for specifying requirements and deriving test cases;
- natural language specifications would still be used for capturing preliminary requirements; the formal description method would be used as a complement for elaborating complicated requirements;
- Nortel software designers and testers would be the eventual users of the chosen method.

4.5.1 Criteria

The problems identified from the interview study represented some of most serious concerns in the investigated Nortel development process. We could view these problems from

\[^1\text{These were assigned according to the opinions obtained from Nortel engineers.}\]
a different perspective and used them as criteria for choosing the best formal description method for tackling those concerns. For instance, difficulties in reflecting specification changes in test cases indicated the need for traceability between artifacts in the requirements and testing phases. Section 2.5 has a discussion of the identified issues. Some of them are modified slightly to fit the context of our evaluation and are shown as follows:

- encourages requirements to be specified completely and clearly,
- helps identify specification problems,
- allows requirements specifiers to produce easily reviewable specifications,
- has testing support, and
- provides traceability between different development artifacts, such as specifications and test cases.

Some of these criteria were already used in the preliminary screening. We intentionally excluded the issue 'more effective use of time', as our limited experience with the methods would not allow an accurate evaluation. Only by applying the selected method to model a larger part of ServiceCreator and keep track of the statistics could we evaluate this issue precisely.

On the other hand, the above list of criteria ignored other important issues, such as usability, that were also crucial for a method to be successfully applied in a commercial setting. When we presented the criteria to the Nortel managers and engineers in a meeting, they elaborated on some of our criteria and added additional ones, such as usability, for ensuring that the chosen method would fit into their development process. Some direct quotes from the minutes of that meeting are shown below:

**Language related**

- describes requirements better than natural language (e.g., more complete, concise, unambiguous),
- provides adequate modeling support for describing system behavior at different level of abstraction
- is useful to both designers and testers
- helps detect problems in the specifications (e.g., inconsistencies and incompleteness),
has an easily reviewable notation that is usable by design and testing groups without much training (preferably usable to less-technical groups such as documentation and training also),

- is a standardized notation with strong industrial support and is applicable to telecommunications systems,
- is executable (e.g., for prototyping), and
- handles large models.

**Tool related**

- is easy to use and has industrial support and acceptance,
- supports parallel development (e.g., multi-user access),
- produces test cases that have better coverage than a manual technique from the specifications,
- provides test execution support, and
- generates code and documentation from specifications.

Based on the same reason for ignoring the 'more effective use of time' issue, we did not have the criteria 'reduces cost' in our list.

Some quotes from the meeting contained multiple vague ideas that must be separated and clarified before we could conduct a precise evaluation. Following the formal description method evaluation frameworks [ACJ+96, DKZ97], we refined the above list and categorized the criteria. Table 4.1 shows the final list that was used to evaluate each method, followed by an explanation of each criterion.

**Language related**

**Usability**

**Readability**—The notation should be easy to use and review. It should have a gentle learning curve so that designers and testers (or even less-technical groups such as marketing and training) can review the specifications without intensive training. For instance, it should not require the readers to have a deep mathematical background.

**Ease of creation**—Similarly, the notation should have a gentle learning curve so that designers and testers can grasp the core concepts of the language
<table>
<thead>
<tr>
<th>Language related</th>
<th>Tool related</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Usability</td>
<td>• Usability</td>
</tr>
<tr>
<td>• Readability</td>
<td>• Checkability</td>
</tr>
<tr>
<td>• Ease of Creation</td>
<td>• Traceability</td>
</tr>
<tr>
<td>• Requirements Support</td>
<td>• Maturity</td>
</tr>
<tr>
<td>• Modeling</td>
<td>• Parallel Development</td>
</tr>
<tr>
<td>• Abstraction</td>
<td>• Artifact Generation</td>
</tr>
<tr>
<td>• Partial Description</td>
<td>• Code Generation</td>
</tr>
<tr>
<td>• Requirements Quality</td>
<td>• Test Case Generation</td>
</tr>
<tr>
<td>• Deeper Understanding of Requirements</td>
<td>• Document Generation</td>
</tr>
<tr>
<td>• Early Detections of Requirements Problems</td>
<td>• Test Case Execution</td>
</tr>
<tr>
<td>• Expressiveness</td>
<td></td>
</tr>
<tr>
<td>• Scalability</td>
<td></td>
</tr>
<tr>
<td>• Maturity</td>
<td></td>
</tr>
<tr>
<td>• Testability</td>
<td></td>
</tr>
<tr>
<td>• Test Specifications</td>
<td></td>
</tr>
<tr>
<td>• Executable Specifications</td>
<td></td>
</tr>
<tr>
<td>• Maintainability</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Formal description method evaluation criteria.

quickly and apply it to specifying system requirements and test specifications, respectively.

**Requirements Support**

**Modeling**—The notation should provide adequate modeling support so that the system requirements can be analyzed as a coherent unit. Requirements specifiers should be able to create a complete high-level description of a system (or a system component) that specifies both normal operating conditions as well as error handling requirements easily. Since requirements for telecommunications applications are mostly control-flow based, the emphasis should be on modeling of the control-flow information. Support for modeling data flow or relations between data is also desirable.

**Abstraction**—Requirements specifiers should be able to specify system requirements in different level of abstraction. For instance, designers should be
able to create a model with enough details for requirements analysis and preliminary design, while testers should be able to hide the internal complexity of a system and analyze only its external behavior for the purpose of generating test cases.

**Partial Description**—Since requirement specifiers are not expected to formalize the entire system, the notation should allow partial formalization. For instance, a designer should be able to formalize just one of the system components and interface this formal model with other existing specifications. The notation should not require the specification to be complete.

**Requirements Quality**

**Deeper Understanding of Requirements**—Requirements expressed in the formal notation should help both the requirements specifiers and the readers understand the system better than if the requirements were written in natural language. For instance, the notation should encourage the requirement specifiers to perform an in-depth analysis and provide a detailed but concise description of the system to the readers.

**Early Detection of Requirements Problems**—The language should guide requirement specifiers to write higher-quality specifications by making requirements problems, such as ambiguities, inconsistencies or incompleteness, more apparent. It should also help reviewers detect these problems easily.

**Expressiveness**—For a given application domain, the notation should be suitable for describing any requirements that may arise in a natural and straightforward manner. For instance, in the telecommunications domain, requirements specifiers should allow properties such as priority, timing, and concurrency to be expressed easily.

**Scalability**—The language should provide facilities for handling large complex specifications effectively. For instance, a notation that supports information hiding or modularity would allow engineers to analyze small parts of a large specification without being affected by its size and complexity.
Maturity—It is preferable that the language be mature and standardized. A large user base and strong industrial support in the telecommunications domain are also important.

Testability

Test Specifications—The language should provide adequate support for testing. For instance, testers should be able to specify test cases as parts of the specifications and use them as a reference during the testing phase. Alternatively, they should be able to extract test information (e.g., external system behavior) from the specifications for ensuring that the implementations would be tested according to the original requirements.

Executable Specifications—The specifications should be executable so that engineers can inspect the effects of the requirements before constructing the actual systems. By animating the specifications interactively, engineers can track errors more easily and obtain additional assurance about the correctness of the requirements.

Maintainability—Engineers should be able to use the specifications as a starting point for the maintenance activities. As requirements are changed over time, it should be easy to reflect the corresponding changes in the requirements. It is preferable that the effects of these changes be minimized and isolated to avoid the need for repeating the analyzes.

Tool related

Usability—The user interface should be intuitive and conform to generally accepted interface standard to minimize the learning effort. The tool should also have full support for those features of the underlying notation that help engineers create high-quality specifications easily.

Checkability—The tool should provide facilities for detecting problems in specifications automatically, which can be simple syntax and consistency checking or advanced verification of system properties. Interactive tools such as simulators or test replayers are also useful.
Traceability—It is crucial that the tool provides traceability between different software development artifacts, such as specifications and test cases. As requirements are changed over time, engineers should be able to identify the affected areas and propagate the corresponding modifications easily.

Maturity—Ideally, the tool should be produced by a company with strong industrial support and a good reputation in the telecommunications domain. It should also be a commercial quality tool with a large user base.

Parallel Development—The tool should support group development, which is crucial in commercial software development. For instance, it should provide multi-user support so engineers could work on different parts of the specifications concurrently.

Artifact Generation

Code Generation—It is desirable that the tool translates specifications to code either for prototyping or for use in the target environment. In the latter case, the generated code should be reasonably efficient and have a clean interface for communicating with surrounding systems.

Test Case Generation—The tool should allow testers to create test cases from specifications, preferably automatically. These test cases should achieve better coverage than those that are created manually. In the case where the tool does not support automatic execution, these generated test cases should be readable by testers.

Document Generation—Engineers should be able to use the tool to generate documents that allow readers to understand the specifications without using the tool. These documents should be created in a report format that can be easily incorporated into existing natural language specifications.

Test Case Execution—The tool should provide built-in facilities for executing the generated test cases and reporting the results automatically.
4.5.2 Limitations

We used a simplified part of ServiceCreator, PasswordCheck, as the subject of our formal description method evaluation. While it did not have the same level of complexity as its larger counterpart, we verified with the Nortel engineers that the essential properties and concepts were similar, and all of our simplifications were minor. While we believe that the component was an adequate reflection of ServiceCreator for most intended purposes, we could only evaluate the scalability of the methods based on whether they provided enough facilities for handling large specifications.

An unavoidable pitfall in our quantitative approach was the possible loss of information in representing our opinions numerically. We needed to assume that we could reflect our opinions in a linear ordinal scale (e.g., a score of 4 in usability is twice as good as a score of 2). While there might not be a complete solution to these problems, the involvement of the engineers in the evaluation process, especially in choosing the criteria weightings, ensured that the scale was appropriate for the purposes of this evaluation. To aid the Nortel engineers in reviewing our evaluation, we also included qualitative justifications with the numerical scores to convey details that were not apparent from the numbers alone.

Last, a confounding factor in feature analysis is the potentially biased opinions of the evaluator. Although we tried to ensure the objectivity of the evaluation, assignments of the scores inevitably contained our subjective judgment, and we did not feel that we could accurately evaluate criteria such as usability. To mitigate this potential problem and to gain more confidence in our assessment, we demonstrated the methods and the models to the Nortel engineers.

4.6 Evaluating Formal Description Methods

In this section, we present brief overviews of the evaluated methods, their notations, and our experience in applying them.
4.6.1 SDT

Tools

SDT from Telelogic is a modeling tool that utilizes the Specification and Description Language (SDL) [CCI88] for behavioral modeling and Message Sequence Charts (MSCs) [IT96] for scenario-based specification. Traceability between these two types of models is provided within the graphical environment. However, the SDL and MSCs parts of the tool can be used independently—users are not required to specify a system in both notations. SDT includes the following tools to ease the task of modeling:

- graphical editors for creating SDL models and MSCs,
- syntax and semantics checkers,
- a symbolic debugger,
- a validator that explores the state space of SDL models automatically to detect application-independent problems such as deadlocks and synchronization problems, and
- a code generator.

There are two notable features in SDT: during a debugging session, the debugger can record the input and output of an SDL model, which essentially lets users derive high-level test cases from the specification; if a user creates some MSCs independently (i.e., not derived from the model), SDT can verify the correctness of the model with respect to those MSCs and provide symbol coverage information. These two features provide a limited form of test case generation and requirements verification capability. The validator can also perform some simple checking of system properties.

Notation

SDL is a standardized (ITU Recommendation Z.100) state-based formal description language that is suitable for describing real-time or stimulus-response systems. It is an executable specification language based on the theory of finite-state machines (FSMs). Models can be specified in either a flowchart-like graphical representation or a less popular textual form.

SDL models have three major layers: the system, block and process layers. Each SDL system contains interconnected blocks, which may be sub-divided into more blocks, and each block may contain multiple processes. Figure 4.1 shows the skeleton of an SDL
model with one block level. Processes communicate with each other and the environment by passing *signals* through *channels*. Because of the hierarchical structure of models, channels may need to pass through several layers before reaching their final destinations, which can either be the environment or other processes. The arrows in Figure 4.1 show the possible placement of the channels.

![Figure 4.1: SDL layers and the possible communications channels.](image)

The core of an SDL model are the SDL *processes* that execute concurrently (see Figure D.3 for an example). All processes start at a stage called *Start*, transitions occur when signals are received, and operations such as variable manipulations are performed during the transitions. Processes are constructed using SDL’s diagramming symbols, show in Figure D.1. Typed variables, such as integer and real as well as mathematical constructs such as sets, can be declared inside each process. A special type called *timer* can also be created to measure time duration.

MSCs are a standardized notation that can be used independently of SDL for specifying interactions between multiple objects over time (see Figure D.4 for an example). However, if traceability between an SDL model and the MSCs is needed, objects in the MSCs must refer to the actual SDL systems, blocks, or processes in the linked model. The instance of each object is represented by a vertical line with time advancing in the downward direction. Synchronous communication between objects is denoted by horizontal arrows going from the timelines of the senders to those of the receivers and are ordered with respect to their occurrence in time.

**Commentary**

SDT has a user-friendly graphical interface that adds no extra complexity to creating and reviewing models. The syntax and semantics checkers are especially useful for detecting
inconsistencies and omissions in the requirements. MSCs and SDL can be used for requirements capturing and modeling, respectively, but testing support is limited as there is no fully automatic test case generation. An interesting feature of SDT is its ability to verify SDL models against MSCs, providing a limited sense of verification without requiring any mathematics from the users. SDT does not natively support multi-user access, but can be integrated with third parties tools to provide this functionality. The debugging and code generation facilities are suitable for producing prototypes.

Both SDL and MSC are graphical notations that allow users to understand and create models easily. We believe that MSCs, which describe the high-level system behavior without exposing the underlying complexity, are intuitive enough that virtually no training is necessary. However, the disadvantage of using MSCs alone is that they lack scalability and do not provide a sense of completeness of the requirements. On the other hand, SDL requires users to understand the concept of FSMs, which can be difficult for non-technical users. Its formal syntax helps clarify unstated assumptions and ambiguities present in natural language specifications. The state-based notation also provides a better sense of completeness to help uncover requirements omissions and allows concurrency and timing requirements to be accurately specified. Although SDL is moderately scalable because of its hierarchical structure and the ability to package a partial SDL model for reuse, the complexity of FSMs grows very fast with respect to the system size. As a result, users are strongly encouraged to model at a proper level of abstraction, or the resulting specifications can be overly complex. Another disadvantage is that complex data relationships cannot be modeled. SDL currently provides limited support for persistent data through the use of variables and axiomatically-defined abstract data types.

The 3-page SDL model and the MSCs created for PasswordCheck took around 4 hours to build (see Appendix D.1). We spent a large amount of time deriving MSCs that tested all scenarios of PasswordCheck. This exercise proved to be advantageous not only to have a complete set of test cases but also to ensure that we built the model correctly. The SDL model created for PasswordCheck was very simple. The translation from the specification to SDL notation was intuitive except for modeling the persistent data. However, we needed to take extreme care in modeling the component at a high level of abstraction to keep the specification small and simple.
4.6.2 TestMaster

Teradyne TestMaster is a test modeling and generation tool that is capable of creating black-box test cases for control-flow and limited data-flow testing. Requirements are specified in a proprietary FSM-based notation that is suitable for describing stimulus-response systems. Since TestMaster operates at the "black-box" level, it cannot be used to model concurrent systems or interactions between multiple system components. The following tools are included in TestMaster:

- a graphical editor for building FSM-based models,
- a test case generator,
- a symbolic debugger, and
- a test replayer for inspecting the behavior of the generated test cases in conjunction with the debugger.

TestMaster creates test cases by exploring paths in the FSM models; these paths are essentially a series of transitions. Each transition is augmented with a fragment of textual information, which can be user-specified test case executor commands for triggering the corresponding transition in the real system or comment for explaining the transition. A complete test case is produced by concatenating the test information of all transitions along a path. TestMaster provides a variety of options to control the coverage of the generated test cases, e.g. transition coverage—all transitions are exercised at least once, or full coverage—all inputs are tested in all states.

Notation

The proprietary and formal notation used in TestMaster describes a system in one large hierarchical FSM (see Figure 4.2). Typed variables can be declared, but the notation does not have the notion of time. The specification consists of a series of states connected by transitions. Some of the states, called model calls, can be refined into sub-FSMs. Figure D.5 shows a top-level FSM in which one of the states, validatePW, is expanded into another FSM shown in Figure D.8. Each transition in the model defines a state change based on external inputs and is defined by some of the following properties:

- event—the name of the event that causes the application to change state,
- predicate—a boolean expression that must be evaluated to be true for a transition to occur,
- constraint—a boolean expression for limiting the amount of generated test cases,
- action—variable assignments that are executed when a transition occurs,
- arguments—parameters that are passed to sub-models, and
- test information—free-form text in this field is concatenated with the test information of other transitions during test case generation to form a test case.

Figure 4.2: The hierarchical structure of an FSM in TestMaster.

Figure D.6 shows some of the commonly used symbols in TestMaster. A model begins at the entry state symbol in the top-level FSM. A state transition occurs if an event occurs or the expression in the predicate field is satisfied. Upon reaching a model call symbol, the control flow is transferred to the lower level FSM and is returned when the exit state symbol is reached.

The major difference between a TestMaster model and other FSM-based models is that it specifies the behavior of a tester, instead of that of the system under test. Since the main purpose of the model is for generating test cases, the specification simply describes the set of possible actions that a tester can take at some particular states and the correct system responses according to the actions taken. Thus, the typical way of modeling behavior of reactive systems—wait for inputs and produce outputs—is reversed. Instead of emphasizing on how different system inputs are handled, TestMaster users focus on specifying the variety of testers' outputs that can be generated. This vastly different perspective makes TestMaster more suitable for describing test cases and less effective for modeling, as designers traditionally concentrate on analyzing system requirements.
Commentary

Although TestMaster has a graphical interface, it is less user-friendly than SDT, as textual information plays an important role in creating the models, debugging them, and creating the test cases. The tool can be used to capture black-box requirements from a tester's perspective, but its main strength is in generating control-flow test cases, which can be extremely useful if non-black-box behavior, timing requirements, and data-flow testing are not important. Also, in cases where the specifications change frequently, test cases can be regenerated without much effort. However, because of the inevitably large number of test cases that must be generated, Nortel testers expressed concerns about placing total trust on an automated tool without having a real feeling of the test coverage. Certainly, the test replayer can be used to inspect each individual test case, but this time-consuming activity may defeat the original purpose of generating test cases automatically. The syntax and semantics checkers in TestMaster help detect inconsistencies and omissions. However, since they detect problems in test specifications rather than requirements specifications, the benefits are limited. Although the tool provides no link between the models and the test cases, a workaround is to put pseudocode in the test information field of all transitions so that the content of the generated test cases would indicate the parts of the model that are being tested.

Also based on a formal FSM notation, the language used in TestMaster shares many of the benefits and disadvantages with SDL. On the other hand, we expect its learning curve to be steeper for testers, who are really the target users. Not only are testers usually less familiar with modeling FSMs, but they must also know the restrictions of the language, such as weak support for persistent data and data-flow testing, in order to use the method effectively. Indeed, during the 10 hours that we took to model Password-Check (see Appendix D.2), our main problem was in incorporating persistent data such as the predefined password and the maximum number of allowable invalid password attempts into the model. After discussing with a Teradyne consultant, we decided that the information need to be encoded directly in the model, reducing the claim of "fully" automatic test case generation to some degree. The resulting model was somewhat inflexible, especially if there was a huge need for data-flow testing. However, generally speaking, we were impressed that an existing commercial tool can provide some form of test case generation without requiring the user to apply mathematical reasoning explicitly.

---

2This concern was raised when we presented our finding of TestMaster to the Nortel testers.
4.6.3 Validator/Req

Validator/Req from Aonix is capable of generating black-box test cases for data-flow testing from requirements specified in UML use case diagrams and sequence diagrams. It includes a graphical editor for creating the diagrams, syntax and consistency checkers, a test case generator, and a document generator. Overall, the tool is very user-friendly mostly because of the simplicity of the notation and the small amount of input required to generate test cases.

The test case generation mechanism in Validator/Req is much simpler than that of TestMaster, mostly because sequence diagrams can be readily used as test cases without much effort. However, for diagrams with parameterized messages (see the message enter-Number in Figure D.12 for an example), Validator/Req creates multiple test cases from each sequence diagram by permuting the parameters in a variety of ways. For instance, if the parameter is an integer, its domain can be defined by specifying a maximum, a minimum and a reference value (e.g., a typical value that the parameter can take). Then, Validator/Req can generate test cases with the parameter set to different data points around the reference and boundary values for testing both valid and invalid cases of the sequence diagrams. Other ways of specifying the valid range are explained in the next section. The number of generated test cases can be altered to achieve different test coverage.

Validator can produce documents that conform to IEEE recommended practice for software requirements specifications [IEE93]. These documents include the following information:

- use case diagrams and sequence diagrams,
- details about the test cases (e.g., descriptions, test execution steps, and pass/fail criteria), and
- a requirements traceability table that shows the relationship between use cases and generated test cases.

Notation

Validator/Req utilizes a subset of the notation of UML's use case diagrams and sequence diagrams for behavioral specifications. The semantics of these diagrams is not fully formalized in the UML standard, but Validator/Req augments the sequence diagrams with necessary formalism to allow consistency checking and test case generation.
In Validator/Req, use case diagrams show the relationship among use cases and between actors and use cases within the system boundary (see Figure D.11 for an example); they are designed to give informal overviews of the system services and the actors that use them. Using the UML terminology [BJR97a], an actor represents a role of an object outside the system who interacts with one or more use cases (e.g., a caller that enters a password to PasswordCheck); a use case denotes a system service as manifested by sequences of messages exchanged between the system and one or more actors (e.g., services that the caller expects PasswordCheck to provide). The relation between any two objects is represented by a line connecting them. Technically, use case diagrams in Validator/Req are built only for organizing the use cases; they do not affect how test cases are generated, and are not subject to any consistency checking.

Each use case may be refined as one or more sequence diagrams that describe the actual interactions among objects over time (see Figure D.12 for an example). Similar to MSCs, each participating object is shown as a vertical timeline with time advancing in the downward direction. Each diagram shows exactly one unconditioned interaction sequence—no loop or branch is allowed. Messages between objects are represented as horizontal arrows going from the timelines of the senders to those of the receivers and are ordered with respect to their occurrence in time; however, precise timing requirements (e.g., a 10-second delay) cannot be specified. Only synchronous messages are supported, but they can be parameterized with variables. Most of the interesting features in Validator/Req, such as consistency checking and test case generation, revolve around these variables, which can be of conventional types such as real, integer, boolean, and string. The valid domains of these variables can be specified in a variety of ways; one of which is mentioned in the previous section. The simplest way is to list all valid values explicitly; regular expressions can be used to define variables that are of string or scalar type; for a string variable, its size and the set of allowable characters can be controlled; discrete points of a scalar variable can also be defined using a resolution value addition to its maximum and minimum. For example, if an integer variable has an inclusive range of 0 to 9 and a resolution of 3, then the valid values are 0, 3, 6, and 9. Clearly, the method that is chosen to specify the domains of the variables affects the type as well as the amount of test cases generated.
Commentary

The simplicity of the notation and the intuitive user interface of Validator/Req allow less technical users to use the tool with minimum training. The tool is suitable for capturing black-box requirements early in the software lifecycle where potential users, system boundary and services are being defined. Validator/Req can generate nicely formatted reports that show details in the specifications and the test cases, as well as the traceability information between them; this functionality allows users to access the results without even using the tool. The process of recording requirements in formalized sequence diagrams help detect ambiguities, but the tool provides little assistance beyond ensuring that data types in parameterized messages are consistently defined and used. The main feature of Validator/Req is the automatic data-flow test case generation. However, the benefits are limited as sequence diagrams are, in a sense, test cases already, and users are required to specify how data is to be tested quite explicitly.

Compared to the notations used in SDT and TestMaster, use case diagrams and sequence diagrams are extremely user-friendly, and it is possible that they can be understood without any training. While these diagrams are ideal for communicating ideas between stakeholders early in the requirements phase, they have limited use for specifying detailed requirements. For instance, describing system behavior using only sequence diagrams provides no sense of completeness. Since the notions of branching and looping are not supported in Validator/Req, it is difficult to visualize the complexity of the systems and how errors are handled. Also, these diagrams do not scale well and lack support for timing requirements, making Validator/Req less attractive for modeling ServiceCreator. Although a modeling tool with better UML support would be more appropriate, we did not find any such tool that utilized a formal notation and yet provided adequate testing support.

Indeed, the advantages and pitfalls of Validator/Req were apparent when we modeled PasswordCheck (see Appendix D.3). Formalizing the natural language specification of PasswordCheck to sequence diagrams was straightforward; we were able to specify most of the requirements in 20 sequence diagrams in three hours. The traceability table in the report was useful for visualizing the link between the test cases and the sequence diagrams, as well as how thorough each diagram was being tested. On the other hand, we also spent comparable amount of time in going through the diagrams and ensuring that all scenarios have been adequately specified. Moreover, since the requirements of
PasswordCheck (and ServiceCreator) contain a lot of control-flow information, we did not find the data-flow test case generation useful, especially when we felt that it was inappropriate to specify the type of data to be tested.

4.6.4 Evaluation

Tables 4.3 to 4.8 show the results of the evaluation. Starting from column one, these tables list the evaluation criteria, the degree the method satisfies the criteria (discussed in Section 4.5.1), and the justifications for the score assignments. Table 4.2 describes the rating scale used in the second column of these tables, listing the descriptions of the six possible ratings in column one and the corresponding numerical scores in column two.

<table>
<thead>
<tr>
<th>Description</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>This criteria is not satisfied because the method provides no support for it.</td>
<td>0</td>
</tr>
<tr>
<td>The method does not satisfy this criteria by itself; “workarounds” or third-party tools are needed.</td>
<td>1</td>
</tr>
<tr>
<td>The method satisfies this criteria only in a very limited sense.</td>
<td>2</td>
</tr>
<tr>
<td>The method satisfies this criteria only in some key aspects.</td>
<td>3</td>
</tr>
<tr>
<td>Except for a few non-essential areas, the method satisfies this criteria strongly.</td>
<td>4</td>
</tr>
<tr>
<td>The method satisfies this criteria completely with all aspects covered.</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 4.2: The 6-point scale for rating the degree a method satisfies a criterion.

Clearly, some of the criteria used in the evaluation are more important than others. When we tried to calculate an aggregated score for each method, we felt that the influence of each criterion in this score should be proportional to its importance to the Nortel engineers. Thus, we used a 4-point scale suggested in [KJ97b] to rate the relative importance of the criteria. Table 4.9 shows the descriptions of the four possible ratings in column one and the corresponding numerical weightings in column two.

Table 4.10 shows a summary of our quantitative evaluation. It lists the evaluation criteria in column one, the weightings that we, together with the Nortel engineers, assigned to the criteria in column two, and the scores there were assigned to SDT, TestMaster, and Validator/Req earlier in columns three, four, and five, respectively. The conclusion of the evaluation was reached by comparing the final scores of the methods that were obtained by summing the weighted scores from each criterion, making SDT the most
suitable tool. We completed the evaluation by demonstrating the methods and the models to the Nortel engineers. They agreed that SDT satisfied their criteria more closely than the other evaluated methods.

4.7 Summary

In the second part of the formal specification pilot study, we conducted a survey to choose a formal description method that was usable in the Nortel development environment, was suitable for modeling telecommunications applications, and could effectively tackle the development problems identified in the interview study.

During the preliminary screening phase, we found that most of the formal description methods we surveyed lacked good modeling and testing support, which was one of our important criteria. After narrowing the search down to three candidates, we used them to formalize a small part of ServiceCreator to learn about their strengths and weaknesses. Applying a set of criteria that was proposed using results from the interview study and opinions from the Nortel engineers, we performed a feature analysis to evaluate the three candidates quantitatively; SDT was chosen as the most suitable method. To overcome some limitations in our evaluation and to gain confidence in the conclusion, we invited the Nortel engineers to be involved in some parts of the evaluation process.
<table>
<thead>
<tr>
<th>Language Criteria</th>
<th>Score</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Readability</td>
<td>4</td>
<td>The graphical notation is very similar to flowcharts, with which most Nortel personnel should be familiar. However, less technical users will need training to acquire some basic knowledge about FSMs. On the other hand, users who concentrate on analyzing external system behavior only need to use MSCs, which is a much simpler notation. Designers should be able to create SDL models without any problems. However, the ability to create FSM-based models requires analytical skills that some testers may not possess. On the other hand, it is expected that testers only need to create MSCs, which should require minimum training.</td>
</tr>
<tr>
<td>• Ease of Creation</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>Requirements Support</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Modeling</td>
<td>5</td>
<td>The hierarchical FSM-based notation provides full support for modeling high-level or detailed system behavior.</td>
</tr>
<tr>
<td>• Abstraction</td>
<td>5</td>
<td>SDL allows users to choose their own level of abstraction—the model can be used for preliminary designs if enough details are included, or black-box requirements specifications if only external behavior is described. MSCs can also be used in the latter case.</td>
</tr>
<tr>
<td>• Partial Description</td>
<td>4</td>
<td>SDL can be used to describe any system component or sub-components; it does not make any assumption about the part of the system being specified, with the restriction that there cannot be any impartiality within the SDL model. However, while users are required to specify all sub-components in the model, at a minimum, they only need to create empty SDL processes for those parts that they do not want to describe in detail.</td>
</tr>
<tr>
<td><strong>Requirements Quality</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Deeper Understanding of</td>
<td>4</td>
<td>The process of formalizing requirements into SDL models encourages requirements specifiers to have a better understanding of the systems, by forcing them to explore the system behavior in detail, clarify the hidden assumptions, and resolve the inconsistencies. However, if the model is not created at a proper level of abstraction, it may contain too much details and confuse the readers of the specifications.</td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Early Detections of Requirements Problems</td>
<td>4</td>
<td>The syntax and the semantics of SDL help reduce ambiguities and inconsistencies. The FSM-based notation also provides a better sense of completeness, helping requirements specifiers detect missing requirements more easily.</td>
</tr>
<tr>
<td><strong>Expressiveness</strong></td>
<td>4</td>
<td>The FSM-based notation is suitable for describing stimulus-response systems. Timing and concurrency are supported. However, requirements about priorities cannot be specified.</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>4</td>
<td>The modular structure of SDL allows large systems to be built incrementally. The ability to package parts of SDL systems for reuse also eases the task of specifying large systems.</td>
</tr>
<tr>
<td><strong>Maturity</strong></td>
<td>5</td>
<td>Since its was first standardized in 1976, SDL has been successfully applied in both the research community and commercial telecommunications sector.</td>
</tr>
<tr>
<td><strong>Testability</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Test Specifications</td>
<td>4</td>
<td>MSCs can be treated as test cases for describing external inputs and expected system outputs easily. However, data-flow information used in the MSCs needs to be explicitly specified.</td>
</tr>
<tr>
<td>• Executable Specifications</td>
<td>5</td>
<td>SDL models can be animated for inspecting the specified system behavior.</td>
</tr>
<tr>
<td><strong>Maintainability</strong></td>
<td>4</td>
<td>The hierarchical and modular structure of SDL models allows requirement specifiers to make small local changes with minimum effects to the entire specification.</td>
</tr>
</tbody>
</table>

Table 4.3: Evaluation of SDT's notation (SDL).
The intuitive user interface allows users to build models that take full advantage of SDL. Visual cues are provided throughout SDT to guide the task of creating the models. However, the online help often does not provide enough information to assist new users.

SDT validates the syntax and semantics of SDL models, detects application-independent problems such as deadlocks automatically, and verifies the correctness of the models with respect to some MSCs. A debugger is also provided for inspecting the models and the MSCs interactively.

SDT provides traceability between SDL models and MSCs within the tool environment. However, because of the sub-par document generation capability, the traceability information is not available in hard-copy format.

The company who produces SDT, Telelogic, is a company with a good reputation in the SDL tool market. This telecommunications industry-oriented tool has been in development for years and has been used by telecommunications companies such as Lucent, Nokia, and Ericsson.

SDT provides no native multi-user support; third-party tools can be integrated to augment the missing functionality.

SDL models can be automatically translated to code for use in the target environment. However, the generated code is useful mostly for embedded systems.

MSCs, which can be used as test cases, need to be manually derived from SDL models. However, assuming that the models are created carefully, the test coverage should be much more complete than manually created test cases. The generated test cases are highly readable and contain detailed information about each input/output pair.

Many useful features of SDT, such as the symbol cross-referencing and the test case traceability, are only available within the tool environment. SDT prints out only the SDL models and MSCs as they appear on the screen.

SDT does not provide built-in facilities for executing test cases in MSC form. However, testers should have no problem in executing these highly readable MSCs. Another tool produced by Telelogic, called iTEX, allows test cases derived from SDL models to be used for automated conformance testing of embedded systems.

<table>
<thead>
<tr>
<th>Tool Criteria</th>
<th>Score</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>4</td>
<td>The intuitive user interface allows users to build models that take full advantage of SDL. Visual cues are provided throughout SDT to guide the task of creating the models. However, the online help often does not provide enough information to assist new users.</td>
</tr>
<tr>
<td>Checkability</td>
<td>5</td>
<td>SDT validates the syntax and semantics of SDL models, detects application-independent problems such as deadlocks automatically, and verifies the correctness of the models with respect to some MSCs. A debugger is also provided for inspecting the models and the MSCs interactively.</td>
</tr>
<tr>
<td>Traceability</td>
<td>3</td>
<td>SDT provides traceability between SDL models and MSCs within the tool environment. However, because of the sub-par document generation capability, the traceability information is not available in hard-copy format.</td>
</tr>
<tr>
<td>Maturity</td>
<td>5</td>
<td>The company who produces SDT, Telelogic, is a company with a good reputation in the SDL tool market. This telecommunications industry-oriented tool has been in development for years and has been used by telecommunications companies such as Lucent, Nokia, and Ericsson.</td>
</tr>
<tr>
<td>Parallel Development</td>
<td>2</td>
<td>SDT provides no native multi-user support; third-party tools can be integrated to augment the missing functionality.</td>
</tr>
<tr>
<td>Artifac Generation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code Generation</td>
<td>4</td>
<td>SDL models can be automatically translated to code for use in the target environment. However, the generated code is useful mostly for embedded systems.</td>
</tr>
<tr>
<td>Test Case Generation</td>
<td>2</td>
<td>MSCs, which can be used as test cases, need to be manually derived from SDL models. However, assuming that the models are created carefully, the test coverage should be much more complete than manually created test cases. The generated test cases are highly readable and contain detailed information about each input/output pair.</td>
</tr>
<tr>
<td>Document Generation</td>
<td>2</td>
<td>Many useful features of SDT, such as the symbol cross-referencing and the test case traceability, are only available within the tool environment. SDT prints out only the SDL models and MSCs as they appear on the screen.</td>
</tr>
<tr>
<td>Test Case Execution</td>
<td>2</td>
<td>SDT does not provide built-in facilities for executing test cases in MSC form. However, testers should have no problem in executing these highly readable MSCs. Another tool produced by Telelogic, called iTEX, allows test cases derived from SDL models to be used for automated conformance testing of embedded systems.</td>
</tr>
</tbody>
</table>

Table 4.4: Evaluation of SDT.
<table>
<thead>
<tr>
<th>Language Criteria</th>
<th>Score</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>● Readability</td>
<td>3</td>
<td>The appearance of the language used in TestMaster is similar to that of FSMs, which should be also easy to understand for most Nortel personnel. However, the descriptions of the transitions, which hold the majority of the information about the models, may contain complicated boolean expressions. Textual information still plays an important role in TestMaster's language, which may cause problems for less technical users.</td>
</tr>
<tr>
<td>● Ease of Creation</td>
<td>3</td>
<td>Other that the fact designers must take a vastly different perspective in the analysis of TestMaster models, the process of creating the models should be straightforward. However, some testers, who are really the target users, are less familiar with modeling system behavior as FSMs. They probably need training to acquire the necessary analytical skills as well as some preliminary knowledge about boolean mathematics.</td>
</tr>
<tr>
<td>Requirements Support</td>
<td></td>
<td></td>
</tr>
<tr>
<td>● Modeling</td>
<td>4</td>
<td>The hierarchical FSM-based notation provides full support for modeling external system behavior, although from a Tester's perspective.</td>
</tr>
<tr>
<td>● Abstraction</td>
<td>3</td>
<td>As the tool is geared toward testers, only black-box requirements can be specified.</td>
</tr>
<tr>
<td>● Partial Description</td>
<td>5</td>
<td>The hierarchical and modular nature of the language allows specifications to be gradually developed. As suggested by a Teradyne consultant, a user can first build a simple FSM and then expand parts of the model by using the model call symbol when there is a need to describe detailed behavior.</td>
</tr>
<tr>
<td>Requirements Quality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>● Deeper Understanding of Requirements</td>
<td>4</td>
<td>Similar to SDL, the process of formalizing black-box requirements into TestMaster models also encourages requirements specifiers to understand the requirements in detail. The specification of only black-box behavior avoids over-complication. On the other hand, non-black-box requirements will not benefit much from the use of the language.</td>
</tr>
<tr>
<td>● Early Detections of Requirements Problems</td>
<td>4</td>
<td>Similar to SDL, TestMaster's language helps reduce ambiguities and inconsistencies and detect missing requirements more easily.</td>
</tr>
<tr>
<td>Expressiveness</td>
<td>2</td>
<td>While the FSM-based notation is suitable for describing stimulus-response systems, the lack of support for timing, concurrency, and priority requirements severely limits its usefulness in expressing detailed behavior of telecommunications applications.</td>
</tr>
<tr>
<td>Scalability</td>
<td>4</td>
<td>The modular structure of TestMaster models allows large systems to be built incrementally. The ability to package parts of sub-FSMs for reuse also eases the task of specifying large specifications.</td>
</tr>
<tr>
<td>Maturity</td>
<td>3</td>
<td>The proprietary language used in TestMaster is relatively young (1995) and ties closely with the tool. The method has been successfully used in some telecommunications companies (see the Maturity criteria under the Tool Criteria category).</td>
</tr>
<tr>
<td>Testability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>● Test Specifications</td>
<td>5</td>
<td>The notation is designed for specifying external system behavior from a tester's perspective. Although the models are not scenario-based, a notion that most testers are familiar with, control-flow as well as limited data-flow information about the test cases can be specified.</td>
</tr>
<tr>
<td>● Executable Specifications</td>
<td>4</td>
<td>TestMaster models are executable. However, the advantages are limited because they are test specifications, not requirements specifications.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>4</td>
<td>Similar to SDL, the hierarchical and modular structure of the language allows requirements specifiers to make small local changes with minimum effects to the entire specification.</td>
</tr>
</tbody>
</table>

Table 4.5: Evaluation of TestMaster's notation.
The user interface of the graphical editor in TestMaster is, in general, user-friendly. However, requirement specifiers are required to enter a lot amount of textual information to describe the transitions. Integrated tools such as the debugger and the test replayer also seem to require too much textual input from the users.

TestMaster validates the syntax and semantics of the specifications automatically. A debugger and a test replayer are also provided for inspecting the models and the test cases, respectively. However, there are only limited benefits in checking test specifications.

TestMaster does not explicitly provide any link between models and test cases. Users are required to perform extra work to obtain the traceability information.

While the tool is relatively new and is based on a proprietary notation, the developer of TestMaster, Teradyne, is a well-established player in the telecommunications equipment testing field. TestMaster has been used by companies such as Lucent [Cla98] and Wildfire [Apf97], who produces speech recognition components for telephony systems.

TestMaster does not provide any multi-user support. However, a separate version-control system may be used to supplement the missing functionality.

TestMaster does not generate any code.

TestMaster generates test cases for control-flow and limited data-flow testing automatically. The amount of generated test cases can be flexibly tuned to achieve different test coverage.

TestMaster models can be printed out. TestMaster also creates a formatted report that lists the generated test cases and models in textual format. However, the report does not provide any traceability information.

TestMaster does not have a built-in test case executor but can be integrated with third-party tools to augment the missing functionality. Also, by entering appropriate information in the test information field of the transitions, the generated test cases can readily be used by any test case executors.

Table 4.6: Evaluation of TestMaster.
<table>
<thead>
<tr>
<th>Language Criteria</th>
<th>Score</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Readability</td>
<td>5</td>
<td>The graphical notation is extremely simple; less technical groups such as marketing should be able to understand the specifications without any training.</td>
</tr>
<tr>
<td>• Ease of Creation</td>
<td>5</td>
<td>Designers and testers should be able to specify behavioral requirements as a sequence of inputs and outputs with virtually no training. This type of behavior specification is especially intuitive to testers.</td>
</tr>
<tr>
<td>Requirements Support</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Modeling</td>
<td>1</td>
<td>The notation provides very little modeling support as system behavior is specified as a group of weakly-related sequence diagrams; it is difficult to express control-flow information or error handling requirements.</td>
</tr>
<tr>
<td>• Abstraction</td>
<td>3</td>
<td>Since the notation is mostly used for specifying preliminary requirements or generating black-box data-flow test cases, only external system behavior can be specified.</td>
</tr>
<tr>
<td>• Partial Description</td>
<td>5</td>
<td>As sequence diagrams are only weakly related, requirements specifiers can describe any system component independently and easily without worrying about the overall completeness.</td>
</tr>
<tr>
<td>Requirements Quality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Deeper Understanding of</td>
<td>3</td>
<td>Since the completeness of the requirements is not enforced, the task of creating sequence diagrams can easily become an activity where the requirements specifiers record what they already know. While formalizing the inputs, outputs, and data used in the specifications provides a slightly better understanding of the requirements, the notation does not encourage the specifiers to analyze the requirements deeper.</td>
</tr>
<tr>
<td>Requirements Problems</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Early Detections of Requirements</td>
<td>3</td>
<td>The process of formalizing requirements as sequence diagrams may help detect ambiguities and clarify hidden assumptions. However, they are less useful in detecting inconsistencies among requirements and missing scenarios, because of the weak links among requirements and the lack of any sense of completeness.</td>
</tr>
<tr>
<td>Expressiveness</td>
<td>2</td>
<td>The scenario-based notation of Validator/Req can be used for describing stimulus-response systems. However, the lack of support for timing, concurrency, and priority requirements limits its usefulness in expressing detailed behavior of telecommunications applications.</td>
</tr>
<tr>
<td>Scalability</td>
<td>2</td>
<td>A large number of sequence diagrams can be difficult to manage, especially when they are long and complicated. Unfortunately, use case diagrams provide only limited ways to organize these sequence diagrams. Validator/Req can be integrated with third-party tools to reduce the scalability problem.</td>
</tr>
<tr>
<td>Maturity</td>
<td>3</td>
<td>The notation used in Validator/Req is a subset of UML, which is a young but highly popular standardized notation, especially in information technology field. However, UML, as well as Validator/Req, is not widely used in the telecommunications domain.</td>
</tr>
<tr>
<td>Testability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Test Specifications</td>
<td>4</td>
<td>Similar to MSCs, sequence diagrams can be conveniently used as test cases for specifying external system behavior. Testers can include data-flow information in the diagrams, but sometimes in a rather awkward manner.</td>
</tr>
<tr>
<td>• Executable Specifications</td>
<td>0</td>
<td>Validator/Req models are not executable.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>4</td>
<td>Sequence diagrams can be changed and maintained independently with minimum effects to other diagrams. However, changes in use case diagrams may induce a large number of changes in the associated sequence diagrams.</td>
</tr>
</tbody>
</table>

Table 4.7: Evaluation of Validator/Req's notation (a subset of UML).
Table 4.8: Evaluation of Validator/Req.

<table>
<thead>
<tr>
<th>Tool Criteria</th>
<th>Score</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>5</td>
<td>The tool is easy to use, mostly because of the simple notation. It is expected that most Nortel personnel are able to use it with minimum training.</td>
</tr>
<tr>
<td>Checkability</td>
<td>2</td>
<td>Validator/Req checks only for syntax and ensures that variables are declared and used consistently.</td>
</tr>
<tr>
<td>Traceability</td>
<td>5</td>
<td>Validator/Req can generate a report that explains the relationship between the sequence diagrams and the generated test cases in detail. This information can also be viewed online.</td>
</tr>
<tr>
<td>Maturity</td>
<td>3</td>
<td>The company who produces Validator/Req, Aonix, is a strong player in the object-oriented modeling tool market. However, Validator/Req is relatively new and has not been widely used in the telecommunications domain.</td>
</tr>
<tr>
<td>Parallel Development</td>
<td>2</td>
<td>Validator/Req provides no native multi-user support, but third-party tools can be integrated to augment the missing functionality.</td>
</tr>
<tr>
<td>Artifact Generation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Code Generation</td>
<td>0</td>
<td>Validator/Req does not generate any code.</td>
</tr>
<tr>
<td>• Test Case Generation</td>
<td>3</td>
<td>Validator/Req generates data-flow, but not control-flow, test cases automatically. Unfortunately, this functionality provides only limited benefits as sequence diagrams are, in a sense, test cases. Moreover, testers also need to specify the type of data that needs to be tested quite explicitly, which can be inflexible.</td>
</tr>
<tr>
<td>• Document Generation</td>
<td>5</td>
<td>Validator/Req generates a report that conforms to the IEEE standard for software requirements requirements. The report contains detailed information about the models and the generated test cases, which allows Nortel personnel to access these artifacts without using the tool.</td>
</tr>
<tr>
<td>Test Case Execution</td>
<td>3</td>
<td>The generated test cases are highly readable and can be used by testers for manual execution. Validator/Req does not provide built-in facilities for automatic execution, but third-party tools can be integrated to supplement the missing functionality.</td>
</tr>
</tbody>
</table>

Table 4.9: The scale for rating the relative importance of the evaluation criteria.

<table>
<thead>
<tr>
<th>Description</th>
<th>Weighting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional, nice to have</td>
<td>1</td>
</tr>
<tr>
<td>Desirable</td>
<td>3</td>
</tr>
<tr>
<td>Highly desirable</td>
<td>6</td>
</tr>
<tr>
<td>Mandatory</td>
<td>10</td>
</tr>
<tr>
<td>Criteria (Notation)</td>
<td>Weighting</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Usability</td>
<td></td>
</tr>
<tr>
<td>• Readability</td>
<td>10</td>
</tr>
<tr>
<td>• Ease of Creation</td>
<td>6</td>
</tr>
<tr>
<td>Requirements Support</td>
<td></td>
</tr>
<tr>
<td>• Modeling</td>
<td>10</td>
</tr>
<tr>
<td>• Abstraction</td>
<td>6</td>
</tr>
<tr>
<td>• Partial Description</td>
<td>10</td>
</tr>
<tr>
<td>Requirements Quality</td>
<td></td>
</tr>
<tr>
<td>• Deeper Understanding of Requirements</td>
<td>6</td>
</tr>
<tr>
<td>• Early Detections of Requirements Problems</td>
<td>6</td>
</tr>
<tr>
<td>Expressiveness</td>
<td>10</td>
</tr>
<tr>
<td>Scalability</td>
<td>6</td>
</tr>
<tr>
<td>Maturity</td>
<td>3</td>
</tr>
<tr>
<td>Testability</td>
<td></td>
</tr>
<tr>
<td>• Test Specifications</td>
<td>10</td>
</tr>
<tr>
<td>• Executable Specifications</td>
<td>3</td>
</tr>
<tr>
<td>Maintainability</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Criteria (Tool)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Checkability</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Traceability</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Maturity</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Parallel Development</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Artifact Generation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Code Generation</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>• Test Case Generation</td>
<td>10</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>• Document Generation</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Test Case Execution</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Weighted Sum</td>
<td>N/A</td>
<td>522</td>
<td>475</td>
<td>435</td>
</tr>
</tbody>
</table>

Table 4.10: Comparison of the three evaluated methods.
Chapter 5

Formal Requirements Modeling

5.1 Overview

After choosing a formal description method and a Nortel software system carefully, we formalized some of the system components using SDL in parallel with the existing development process and kept track of a variety of productivity data for comparison. Overall, we were able to show that the use of SDL shortened the development cycle while improving the software quality. However, limitations of the environment of the study prevented us from obtaining some essential metrics, which affected the accuracy of our result.

We explain the method of the formalization process in Section 5.2. In Section 5.3, we show the SDL model and the test cases that were created for the ThruDial component. The findings are presented in Section 5.4, and Section 5.5 summarizes this chapter.

5.2 Method

In this phase, we used SDL to formalize some parts of ServiceCreator. Our goals were to

- identify the technical concerns in formalizing ServiceCreator,
- observe the qualitative effects of applying SDL, and more importantly,
- measure the change in software quality and development timeframe.

When we began our formalization process, the development of ServiceCreator was in the testing phase: the requirements specification went through five manual reviews, the design and implementation were completed, and the Nortel testers were testing the system as the designers were rectifying the reported errors. Unfortunately, due to the
exploratory nature of our study, the Nortel personnel did not allow us to create a controlled environment where we could investigate the effect of formalization on the entire development process. For instance, we did not use SDL throughout the entire development lifecycle, in particular, during the preliminary requirements analysis\(^1\). Rather, starting from the inspected requirements specification, we modeled the run-time behavior of 15 voice-service components of ServiceCreator in parallel with the existing development process (see Figure 5.1). This setting was compatible with the expected use of SDL models where only certain requirements would be selectively modeled for complementing the natural-language specification, as mentioned in Section 4.5. The act of formalizing an existing specification was also similar to performing a rigorous requirements inspection. Afterwards, we derived test cases from the formal model using SDT. We then applied these test cases to the first version of the implementation that was available to testers for official testing.

![Figure 5.1: The formalization process and the existing development process.](image)

During the formalization process, we also kept track of statistics such as

- the time to model the ServiceCreator components,
- the number of errors found in the ServiceCreator specification,
- the number of test cases created from the SDL model,

\(^1\)Actually, SDL is not suitable for the early requirements phase.
the time to create these test cases, and
- the number of implementation errors identified using these test cases

for comparison with similar data from the existing development process, provided by the Nortel engineers. By keeping track of these data, we assumed that the impact of the formalization would have been felt in the inspection, design, implementation, and testing phases. Section 5.4 provides details on the collected data and problems in analyzing them. Some of the productivity information is aggregated to measure the effect of the formalization on the length of the development cycle. No code-related statistics (e.g., implementation defects per line of code) were collected—ServiceCreator was created in a high-level 4th generation language (4GL), and it was difficult to judge its size and complexity by the number of lines of code. Besides, there would be no basis for comparison as our SDL model was not used to create another implementation. Post-delivery information was also not collected because the system was still not released to the customers at the time this study was conducted.

Throughout the study, we also made qualitative observations on the use of SDL, such as the technical problems we encountered, the strengths and weaknesses of SDL, and how we perceived SDL could be used by the Nortel engineers to improve their development process.

5.3 Formal Modeling

The 15 voice-service components of ServiceCreator were modeled as a 70-page SDL system in which the environment contained the underlying OA&M software and messaging-system hardware. Readers are encouraged to consult Section 3.3 and Appendix C for brief overviews on ServiceCreator and SDL, respectively. Of the 15 voice-service components, 10 were modeled as separate SDL processes. The functionality of the five remaining components (e.g., start and end) was incorporated into the main process that administers the control-flow of the telephony application being modeled.

To provide a detailed illustration, we present the SDL model of a telephony application with one ThruDial component and one PasswordCheck component, with the focus on explaining the former in the remaining sections of this chapter. Section 3.3.2 provides a high-level description of these components, and their original specifications can be found in Appendix B.2. Unfortunately, these specifications, as well as the ones for other
voice-service components, lacked a lot of information necessary to model the components accurately. The missing assumptions for ThruDial are presented in Section 5.3.6. Also, to simplify our discussion, only signals, data types, SDL blocks, and SDL processes relevant to ThruDial and PasswordCheck are shown in the SDL models. For instance, we removed all references to components such as FaxSelect, FaxSend, and Menu.

5.3.1 Naming Conventions

In the SDL model, we named the signals and constants using the nomenclature that provides hints about their purpose. We did not use any special convention for naming other constructs such as channels and SDL processes. However, we capitalized all SDL keywords in our model for clarity.

Signals. There are three types of signals in the model: input signals (e.g., input from a caller), output signals (e.g., a voice prompt that ThruDial plays), and internal signals (e.g., inter-process communication). All names of the first signal type are postfixed with \_I, indicating that they are input signals with respect to ServiceCreator. For instance, we modeled the action of pressing the * key on a telephone using a signal named asterisk\_I. Similarly, the names of the the second signal type are postfixed with \_O. The names of internal signals do not have any prefixes or postfixes.

Constants. In our model, constants, or synonyms in SDL terminology, are used to represent values that do not change during the run-time execution of a telephony application, such as the system default greeting prompt. All names of the constants used in the model are in block letters. Also, constants related to a particular component are prefixed with xx where xx is a two-letter acronym representing a particular voice-service component. For instance, we denoted one of the help prompts that ThruDial plays using the constant TD\_HELP\_NAMEEMPTY.

5.3.2 A ServiceCreator Application

Figure 5.2 shows the high-level overview of the SDL model. It has a system called messagingSystem that contains a block called SCAplication, representing a ServiceCreator telephony application. In the future, other components of the messaging system such as a user name database and a call-processing subsystem can be modeled as SDL blocks.
and added to this SDL system. The block *SCApplication* contains an SDL process, called *main*, which administers control-flow of the telephony application, i.e. activates the appropriate voice-service component when a caller exits via an output path. The procedure *signalRedirect* routes signals from the environment to the active component, and vice versa.

The voice-service components are modeled as SDL processes that reside in an SDL package called *Common*. Inside an SDL package, a process is called a process type. This package construct allows the reuse of SDL processes by different SDL models. Thus, we can build models for various ServiceCreator applications without any modification to the SDL processes of the voice-service components. The separation of logic in these components and that in the ServiceCreator applications also eases the task of maintaining the specification. We only show only two SDL processes in this package, but the expanded version used in our study has 10 processes.

![Diagram](image)

**Figure 5.2: A high-level overview of various constructs in the SDL model.**

Figure 5.3 shows the SDL system *messagingSystem*. The keyword *Use Common* in the top-left corner indicates that the system uses the package *Common*. The channel *fromCaller* passes inputs from a caller to the SDL block *SCApplication*, and the channel *toCaller* passes signals from the block back to the caller; the actual signals are represented by *(callerInput)* and *(sysOutput)*. These are *signallists*, which is an SDL notation for representing a collection of signals. Similarly, the signal lists *SC2Underlying* and *underlying2SC* are the signals that pass through the channels *toUnderlying* and *toSC*, respectively. These two channels let the ServiceCreator application communicate with the underlying components of the messaging system.

In Figure 5.4, keywords *Signal* and *Signallist* are used to declare signals and signal lists, respectively. The description of the signals is presented in Table 5.1. In the expanded version of the SDL model, there are 23 signals, eight of which are external (used
for communicating with the environment) and the rest internal.

For parameterized signals, data types of their parameters are shown inside the brackets following their names. For instance, the signal `retrieveName_O` carries a parameter of the type `NumericString`—a string type that can only hold numerical characters and empty spaces. User-defined data types, `singleDigit`, `positiveInt`, `numStrArray`, and `globalStateS` are shown in Figure 5.5. The data type `singleDigit` is an integer type with a range from 0 to 9. Similarly, the type `positiveInt` restricts the integer domain to non-negative numbers. The type `numStrArray` is an integer-index string array, and the type `globalStateS` is a C-like record type holding persistent data that the application keeps track of during run-time. The first and the second data members denote the number of invalid password attempts in the call session and the maximum number allowed, respectively. The member `isTouchTone` indicates whether the caller is assumed to be a rotary caller, and the member `nameDialPfx` is the name-dialing prefix of the ThruDial component. In the expanded version of the SDL model, there are 16 user-defined data types and 20 data members in `globalStateS` type.

Figure 5.6 shows the content of the SDL block `SCApplication`. The process `main` acts as a gateway between the voice-service components and the caller. It re-routes inputs from and outputs to the caller using the procedure `signalRedirect`. The reason for having a router instead of direct connections between different components and the caller is to simplify the input/output interface of the block `SCApplication`. On the other hand, processes of voice-service components exchange signals directly with the underlying
Signal number_I(singleDigit), asterisk_I, pound_I, onHook_I, voice_0(Charstring);

Signallist callerInput = number_I, asterisk_I, pound_I, onHook_I, voice_0;
Signallist SCOutput = voice_0;

Signal done(Integer, globalStateS), ackName_I(Integer, numStrArray),
ackDN_I(Integer), retrieveName_0(NumericString),
chkStatus_0(NumericString);

Signallist underlying2SC = ackDN_I, ackName_I;
Signallist SC2Underlying = retrieveName_0, chkStatus_0;

Figure 5.4: Signal declarations.

Syntype singleDigit=Integer
  Constants 0:9
Endsyntype;

Syntype positiveInt=Integer
  Constants >=0
Endsyntype;

Newtype numStrArray Array(Integer, NumericString)
Endnewtype;

Newtype globalStateS Struct
  curTotalPWRetry,
  maxTotalPWRetry positiveInt;
  isTouchTone Boolean;
  nameDialPfx NumericString;
Endnewtype;

Figure 5.5: SDL data type declarations.

subsystems since only a few ServiceCreator components have this type of communication. There are only two processes in the diagram; however, in the expanded version of the SDL model, the process main has ten processes surrounding it. The other purpose of this process is to administer the control-flow of the ServiceCreator application being modeled, and its logic, shown in Figure 5.7, reflects this. Figure 5.8 shows the graphical view of the application being modeled.

Initially, the process main initializes persistent data in the variable globalState and then invokes the SDL process for ThruDial asynchronously; the configuration of this component is passed as a set of parameters. For instance, information such as the persistent data (globalState), the initial greeting prompt (promptMethod), the dialing mode (is-
ByNumber, isByName, and isByNameNumber), whether to allow only fixed-length phone numbers (isFixedNum) and the maximum allowable length (fixedLen), and whether to left-pad the caller's entry and the digit string to be padded (isLeftPad and leftPad). In the actual implementation of ServiceCreator, the GUI dialog of ThruDial validates these parameters. However, since we formalized only the run-time behavior in our study, we needed to ensure that these parameters were configured correctly in the SDL process. For instance, one of the dialing modes have to be selected, and the variable leftPad must be non-empty if the variable isLeftPad is set to be true.

After invoking the ThruDial process, the process main executes the procedure signalRedirect to redirect the communication between the callers and the ServiceCreator application. When the caller leaves the component, this procedure ends, and the variable exitCode indicates the output path taken. For ThruDial, this variable can take on the following values:
The constants with the prefix C. are exit codes that are common to all voice-service components, while those with the prefix TD. are specific to ThruDial. Different actions are taken according to this code. As shown in Figure 5.7, the SDL process PasswordCheck is invoked if the caller leaves via the # (Cancel) output; the call session terminates if other paths are taken. The invocation of the process PasswordCheck is similar to that of the process ThruDial.

We had to hard-code the parameters of the two components and flow of control of the application shown in Figure 5.8 in the SDL process main. As a result, we needed different main processes to model ServiceCreator applications with different components or control-flow logic. SDT provides facilities for switching between the various main processes conveniently.
Figure 5.7: The SDL process main.

Figure 5.8: The graphical view of the ServiceCreator application illustrated in Figure 5.7.
### Table 5.1: Descriptions of the SDL signals.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>number.I</td>
<td>It models an action of pressing a digit key on a telephone.</td>
<td>The parameter indicates the digit key that the caller enters.</td>
</tr>
<tr>
<td>asterisk.I</td>
<td>It models an action of pressing the * key.</td>
<td>N/A</td>
</tr>
<tr>
<td>pound.I</td>
<td>It models an action of pressing the # key.</td>
<td>N/A</td>
</tr>
<tr>
<td>onHook.I</td>
<td>It models an action of hanging up a call.</td>
<td>N/A</td>
</tr>
<tr>
<td>voice.O</td>
<td>It models an action of playing a voice prompt.</td>
<td>Instead of representing one signal for each prompt, we used a parameterized signal to minimize the number of output signals.</td>
</tr>
<tr>
<td>retrieveName.O</td>
<td>When ThruDial is in the name-dialing mode, this signal is sent by the application to the underlying component in the messaging system to retrieve phone numbers that match the caller input.</td>
<td>The parameter carries the caller input.</td>
</tr>
<tr>
<td>ackName.I</td>
<td>In response to the signal retrieveName, an underlying component in the messaging system sends this signal to answer the query.</td>
<td>The first parameter contains the number of matches, and the second parameter carries the list of phone numbers that matching the caller input.</td>
</tr>
<tr>
<td>chkDNStatus.O</td>
<td>The application sends this signal to an underlying component in the messaging system to check the status of a DN (e.g., free or busy).</td>
<td>The parameter contains the DN to be checked.</td>
</tr>
<tr>
<td>ackDNStatus.I</td>
<td>In response to the signal chkDNStatus, an underlying component in the messaging system sends this signal to answer the query.</td>
<td>The parameter indicates the status of the DN.</td>
</tr>
<tr>
<td>done</td>
<td>An SDL process that models a voice-service component sends this internal signal to the process main when a caller leaves a component.</td>
<td>The first parameter denotes the output path that the caller takes. The second parameter, globalStates, carries persistent data that the ServiceCreator application keeps track of.</td>
</tr>
</tbody>
</table>
5.3.3 ThruDial

We modeled ThruDial as a 9-page SDL process, shown in Figures 5.10 to 5.18. In creating these diagrams, we emphasized clarity over compactness. For instance, there are many cases where we duplicated parts of the diagrams to separate dissimilar ideas clearly, even though the logic required to model these ideas may be extremely similar (see Figure 5.15 for an example).

In Figure 5.10, several local variables are first initialized. Also, if the "name or number" dialing mode is selected, some of the parameters are modified so that the initial dialing mode becomes variable-length number dialing. A greeting prompt is then played unless ThruDial is configured to skip the greeting. The action of playing this prompt is abstracted as sending the signal voice.O to the environment. If a system-default prompt is selected, ThruDial plays one of the four different prompts, TD.GTG_NAME, TD.GTG.VARNUM, TD.GTG.FIXNUM, or TD.GTG.NAMENUM, depending on the dialing mode. These string constants are used as the parameter for the signal voice.O. The values inside these constants are not the actual content of the prompt. Rather, they are used to explain the purpose of the prompt (see Figure 5.9 for an example). This allows the actual prompts to be changed, which happens frequently during the development of ServiceCreator, without affecting the SDL model. The SDL process ThruDial has a total of 20 such prompts.

SYNONYM
TD.GTG_NAME = 'The greeting prompt for ThruDial in the name-dialing mode.';

Figure 5.9: The definition of the constant TD.GTG_NAME.

Figure 5.11 shows the main state waitTimeOut where the process waits for a caller's input. The timer idleTimer is set to activate after an arbitrary amount of time for detecting idleness. In the original specification, there are two types of timeouts: a shorter timeout for rotary callers and a longer one for touch-tone callers, both of which are abstracted into one timer for simplicity. ThruDial assumes that a caller is using a rotary phone if the data member isTouchTone in the persistent data variable globalState is false. If a timeout occurs for a rotary caller, the process sends a signal done with the first parameter set to C.ROTARY before it terminates, indicating that the rotary caller leaves the component because of a timeout. However, for touch-tone callers, different actions are taken according to the dialing mode, as shown in Figure 5.13. For name dialing, the process memorizes that one more timeout has occurred (see the connector
incTOCount on the right side). For number dialing, if the process discovers that the caller has entered a single “0”, it sends the signal *done with the first parameter set to *TD_ATTENDANT and terminates, indicating that the caller leaves the component using the 0 (Attendant) output path. For fixed-length number dialing with left-padding, an extra step is performed to check if the concatenation of the caller entry and the padding string meets the fixed-length requirement (see the bottom-right part of Figure 5.13). The process attempts to transfer the caller to the padded directory number (DN) if there are enough digits in this number. In Figure 5.14, the procedure fillLeftPadding illustrates how the caller entry is padded. In all other number-dialing modes, the timeout sequence is the same as that of name dialing. If the number of timeouts exceeds the maximum defined in *TD_MAXTO, the process sends the signal *done with the parameter set to *TD_NORESP and terminates. This models the behavior where the caller leaves ThruDial via the *No Response output.

In the state *waitTimeout of Figure 5.11, the process may receive a signal *asterisk., which indicates that the caller presses the * key. In name-dialing mode, one of the prompts, *TD_HELP_NAMEEMPTY or *TD_HELPNAMECONTENT, is played depending on whether the caller has keyed in some digits. For number-dialing mode, the prompt *TD_HELP_NUMEMPTY is played if there has been no input from the caller. Otherwise, a pause, which is abstracted as a space character, is added to the caller entry.

If the caller enters a digit key, the process receives a signal *number., with the parameter set to the digit pressed. All received digits are concatenated and stored in the variable *numRecv. Then, for the “name or number” dialing mode, the process checks if the received digits match the name dialing prefix *nameDialPfz (see the top-right part of Figure 5.12). If a match is found, the process switches to name dialing and lets the caller re-enter another DN. In the variable-length number dialing mode, the process transfers the caller automatically if the number of received digits equals *TD_MAXDIGIT, which has a value of 15. In the fixed-length number dialing mode, if the number of received digits reaches the fixed length defined in *fixedLen, the scenario is identical. However, for name dialing (see the bottom half of Figure 5.12), the process checks for a possible request for an attendant. Then, it attempts a name match using the existing entries in *numRecv by sending the signal *retrieveName., which models the action of sending a query to an underlying subsystem (e.g., a user name database) in the messaging system. The process terminates the name entry and asks the caller to enter another name if no match is found, or transfers the caller if there is a single match. Otherwise, the process
waits for more inputs from the caller.

If the process receives the signal \textit{pound.I} at the state \textit{waitTimeout} of Figure 5.11, it sends the signal \textit{done} with the parameter set to \textit{TD\_CANCEL} if the caller has not previously entered any digit (see the top part of Figure 5.15); this models the behavior where the caller leaves the component using the \# (Cancel) path. Otherwise, the processing of this signal \textit{pound.I} differs according to the dialing mode. For number dialing, the process first checks if the caller enters "0#", which is an attendant request. For fixed-length number dialing, the process performs an extra step to ensure that the entered digits meet the fixed-length requirement before transferring the caller (see the bottom part of Figure 5.15). If there are not enough digits, the process plays the error prompt \textit{TD\_PPT\_TOOSHORT} and prompts the caller for another DN.

The case for name dialing is slightly more complicated. Similar to its timeout scenario, the process sends the signal \textit{retrieveName.O} to query for a list of matching names and the corresponding DNs using the caller entry (see the top part of Figure 5.16). The process retrieves the result from the parameters of the signal \textit{ackName.I}. If there is zero or one match, the behavior is the same as the timeout scenario. However, when there are too many matches (i.e., more than \textit{TD\_MAXNAME}, which has a value of nine), the system plays an error prompt \textit{TD\_PPT\_2MANY}, and the caller is required to re-enter another DN. According to the specification, if the number of matches is between two and nine, ThruDial plays a prompt that contains a list of matched names. In our model, we abstracted this prompt using \textit{TD\_PPT\_NAMELIST} with the number of matches appended to its end. The process then proceeds to the state \textit{waitSelName} where it waits for the caller to select one of the names (see the top part of Figure 5.17). Unfortunately, we needed to make assumptions about this state, as the original specification only provide an inadequate description. For instance, we assumed that if

- the process receives the signal \textit{asterisk.I}, it repeats the name listing prompt,
- the process receives the signal \textit{pound.I}, it resets the caller entry and prompts her to enter a new DN, and
- the caller stays idle for too long, the process plays the prompt \textit{TD\_PPT\_TO1} for the first timeout, and \textit{TD\_PPT\_TO2} for the second timeout. The process sends the signal \textit{done} with the parameter set to \textit{TD\_NORESP} and terminates if another timeout occurs again, modeling the behavior where the caller leaves via the \textit{No Response} path.
If the caller enters an invalid name selection, the process plays the error prompt \textit{TD.
INVALIDKEY}, which encourages the caller to make a valid choice. For instance, if there are five matches, digit keys from "1" to "5" are valid. When a valid selection is made, the SDL process transfers the caller to the DN associated with the selection. This information is obtained earlier from the parameters in the signal \textit{ackName.I} (see the top part of Figure 5.16). The processing prior to the actual transfer is modeled in Figure 5.18. First, the signal \textit{checkDNStatus.O} is sent to query the status of the DN, and the result is obtained from the parameter of the signal \textit{ackDNStatus.I}. The process behaves differently according to the value in this parameter:

\begin{itemize}
  \item \textit{TD.IDLEDN}\quad The DN is valid and free. The process sends the signal \textit{done} with the parameter set to \textit{TD.SUCCESS} and terminates, indicating that the caller leaves ThruDial via the \textit{Success} path;
  \item \textit{TD.BUSYDN}\quad The DN is valid but busy. The process informs the caller about the problem by playing an error prompt \textit{TD.PPT.BUSYDN}. Then, it sends the signal \textit{done} with the parameter set to \textit{TD.SUCCESS}, indicating that the caller leaves ThruDial via the \textit{Success} path;
  \item \textit{TD.XFERFAIL}\quad The DN is valid, but there is a hardware call transfer problem. The process informs the caller about this problem by playing an error prompt \textit{TD.PPT.XFERFAIL} and lets her re-enter another DN;
  \item \textit{TD.MBEXISTS}\quad The DN, which has an associated mailbox, is invalid. The process sends the signal \textit{done} with the parameter set to \textit{TD.SUCCESS} and terminates, indicating that the caller leaves ThruDial via the \textit{Success} paths;
  \item \textit{TD.NOMB}\quad The DN, which has no associated mailbox, is invalid. The process informs the caller about this problem by playing an error prompt \textit{TD.PPT.NOMB} and lets her re-enter another DN;
  \item \textit{TD.RESTRICTDN}\quad The DN is blocked by the permission/restriction list. The process informs the caller about this problem by playing an error prompt \textit{TD.PPT.RESTRICTION} and lets her re-enter another DN.
\end{itemize}

The ThruDial component does not perform the actual call transfer; it simply redirects the caller to another subsystem in the messaging system, which may transfer her to the specified DN or the associated mailbox. However, from the perspective of ThruDial, the call session ends after the redirection.
Figure 5.10: Page 1 of the SDL process *ThruDial*. 
Process Type ThruDial

- FPAR
  globalState globalStateS,
  promptMethod CharString,
  isByNumber, isByName, isByNum Booleans
  isFixedNum, isLeftPad Booleans,
  fixedLen Integer,
  leftPad NumericString;

- DCL
  inputDigit singleDigit,
tmpChar NumericString;

- a digit key is pressed
  number_i
    (inputDigit)
    (inputDigit)
    numTO := 0
    globalStatetaIsTouchTone := true
    Reset (idleTimer)
    int2Str(inputDigit, tmpChar)
    numRecv := numRecv // tmpChar
    numRecvLen := numRecvLen + 1
    digitRecv

- the * key is pressed
  asterisk_i
    idleTimer
    numTO := 0
    globalStatetaIsTouchTone := true
    Reset (idleTimer)
    the # key is pressed
    pound_i
    numTO := 0
    globalStatetaIsTouchTone := true
    if done (C_ROTARYDIAL globalState) then
      true
    else
      delayTO
done
    end
    Reset (idleTimer)
    pound

- the caller never presses any key in the call session;
  assume that she is a rotary caller
  the caller never presses any key in the call session;
  assume that she is a rotary caller
  if the key " is not pressed as the first digit in number dialing,
  it is treated as a pause;
  we use a space character to represent this
  if the key " is not pressed as the first digit in number dialing,
  it is treated as a pause;
  we use a space character to represent this

- voice_O (TD_HELP_NAMEEMPTY)
  voice_O (TD_HELP_NAMECONTENT)
  voice_O (TD_HELP_NAMECONTENT)
  voice_O (TD_HELP_NAMECONTENT)
  voice_O (TD_HELP_NAMECONTENT)
  voice_O (TD_HELP_NAMECONTENT)
  voice_O (TD_HELP_NAMECONTENT)

Figure 5.11: Page 2 of the SDL process ThruDial.
Figure 5.12: Page 3 of the SDL process ThruDial.
Figure 5.13: Page 4 of the SDL process ThruDial.
Figure 5.14: The procedure *fillLeftPadding* inside the SDL process *ThruDial*.
Figure 5.15: Page 5 of the SDL process *ThruDial*.
Figure 5.16: Page 6 of the SDL process *ThruDial*. 
Figure 5.17: Page 7 of the SDL process ThruDial.
Figure 5.18: Page 8 of the SDL process *ThruDial*. 
5.3.4 Level of Abstraction

The entire modeling process was relatively straightforward as we encountered no major problems in modeling the voice-service components of ServiceCreator; we also felt that a background in FDTs was not required. However, the biggest concern was to determine the appropriate level of abstraction which was dictated by two opposing needs: the model should be constructed from a black-box view to reduce its complexity, while the exact behavior of the system needs to be modeled for deriving sufficiently detailed test cases. In addition, a more detailed model would help in identifying problems in the natural-language specification.

Our approach was to start from a high level of abstraction, filling in details about some parts of the behavior only if the natural-language specification required it. For example, in modeling ThruDial, we represented the two types of timeouts by just one SDL timer *idleTimer* (see the top part of Figure 5.11), as their actual lengths were relatively unimportant, and in no cases did they ever need to be active simultaneously. We also did not include the actual content of the voice prompts in the model, as mentioned earlier. In addition, we did not model the fact that ThruDial prompts are interruptible. We felt that this trivial behavior is already fully understood by the designers and the testers. Yet, to incorporate this requirement to our model, we needed to add one extra state every time ThruDial plays a voice prompt (see Figure 5.19 for an illustration), which would complicate the model significantly without adding much insight.

![Figure 5.19: Two possible ways to model the action of playing a voice prompt: a) a simplified approach used in the study, and b) a more accurate approach.](image)

As illustrated in the previous section, we modeled the run-time behavior of only the voice-service components and abstracted that of other underlying subsystems in the
messaging system (e.g., the user name database) away as the environment. This simplified the SDL specification and allowed us to concentrate on modeling the core functionality; the model also corresponded closer to the original specification. Clearly, the model did not represent the complete system behavior, e.g., name matching was not part of our ThruDial model. Thus, if testers intended to use the test cases derived from this formal model for testing the implementations, they needed to fill in the missing information. On the other hand, we structured our model so that if the need to specify complete system behavior becomes important in the future, the missing subsystems can be modeled as additional SDL blocks with virtually no changes to the rest of the specification.

Last, we tried to simplify the specification further by modeling only voice-service components that were worth extra attentions (i.e., complicated or particularly important), leaving the conceptually simple components in their original natural-language form. While these components could also benefit from the formalization process, the leverage would be much smaller, since designers could probably describe these requirements informally and still fully understand them. Not only did the omission of these components increase the efficiency of the entire formalization process, but the resulting model also became smaller.

Processing of user input, on the other hand, required reasoning on the level of single digits. In our model, digits received from the caller were actually concatenated and stored in a variable numberRecv without much abstraction; such detail is necessarily to model behavior such as the timeout sequence, the help request, and the name selection accurately. While this treatment could potentially lead to large and cluttered models, we sometimes had to resort to it to be able to derive sufficiently detailed test cases.

5.3.5 Test Case Derivation

To obtain immediate benefits from the formalization process, 120 MSCs were derived from the 70-page SDL model for testing the implementation. The derivation was not automatic: SDT recorded the interactions between the SDL system and its environment as MSCs when we animated the SDL model manually. We felt that the automation was not necessarily desirable since this exercise gave us confidence in the content and the coverage of the test cases.

During the test case derivation, we took advantage of the modular nature of the voice-service components and generated test cases for each of them separately, achieving
full transition coverage in the corresponding SDL processes. Essentially, we created one ServiceCreator application model, such as the one shown in Figure 5.20, to test each component. In this Figure, the ThruDial component is the test subject, and three Announcement components are for identifying the output path taken by the caller. We created 23 MSCs for testing ThruDial, and their content is briefly described below.

![Figure 5.20: The graphical view of the application created for testing ThruDial.](image)

**Test Cases for the Name Dialing Mode**

1. Use the default greeting prompt. Test the scenarios where there are zero and multiple name matches (i.e., more than two but less than ten). Ensure that the caller entry is reset after the first scenario. For the name-selection state in the second scenario, make an invalid selection before selecting a valid name. Upon leaving ThruDial, test the case where the DN is valid and free.

2. Use an empty greeting prompt. Test the scenarios where there are excessive name matches (i.e. more than nine) by pressing the # key as well as wait for the timeout sequence. Then, key in some digit entries to test the scenario where there is a single name match. Upon leaving ThruDial, test the case where the DN is busy.

3. Use a user-defined greeting prompt. Go to the state where the caller can make a selection from a list of names. Timeout twice, and then press the * key to retrieve the help prompt. Timeout twice again to ensure that the previous timeout count is cleared. Enter an invalid name selection, and then timeout twice to perform the same check. Finally, test the full timeout sequence.

4. Go to the name-selection state. Press the # key to restart the name entry and then press the # key again to leave ThruDial.

5. Enter the * key as the first entry to retrieve a help prompt. Enter some digits, and then press the * key again to get a different help prompt.
6. Press the 0 key as the first entry to leave ThruDial.

**Test Cases for the Fixed-length Number Dialing Mode**

7. Use the default greeting prompt. Set the fixed length to be five and the left-padding string to be a three-digit number. During the call session, press the # key to leave ThruDial.

8. Set the fixed length to be five and the left-padding string to be a three-digit number. During the call session, enter a two-digit number. Then, test the scenarios where the caller presses the # key as well as waits for a timeout. Upon leaving ThruDial, test the case where there is a hardware call transfer failure.

9. Set the fixed length to be five and the left-padding string to be a three-digit number. During the call session, enter four digits. Then, test the scenarios where the callers press the # key as well as waits for a timeout. Upon leaving ThruDial, test the case where the DN, which does not have an associated mailbox, is invalid.

10. Set the fixed length to be five and the left-padding string to be a three-digit number. During the call session, enter a single digit. Then, test the scenarios where the caller presses the # key as well as waits for a timeout. Afterwards, when ThruDial prompts for another DN, enter a five-digit number to invoke the call transfer. Upon leaving ThruDial, test the case where the DN, which has an associated mailbox, is invalid.

11. Set the fixed length to be five, and the left-padding string to be a five-digit number. Test the scenarios where the caller waits for a timeout as well as presses the # key without entering any input prior to that.

12. Set the fixed length to be five, and the left-padding string to be a three-digit number. During the call session, enter a six-digit number.

13. Use the fixed-length number dialing greeting prompt. Set the fixed length to be five and the left-padding string to be a six-digit number. Enter a single digit. Then, test the scenarios where the caller presses the # key as well as waits for a timeout.

**Test Cases for the Variable-length Number Dialing Mode**

14. Use the default greeting prompt. Enter more than 15 digits, and then wait for ThruDial to perform the call transfer.

15. Enter the * key as the first entry to retrieve a help prompt. Key in a DN with
some pauses in between, using the * key. Then, test the scenarios where the caller presses the # key as well as waits for a timeout. Upon leaving ThruDial, test the case where the DN is restricted.

16. Enter the 0 key. Then, test the scenarios where the caller presses the # key as well as waits for a timeout to leave ThruDial.

**Test Cases for the “Name or Number” Dialing Mode**

17. Enter some digits and press the # key. Upon leaving ThruDial, test the case where the prefix of the DN is restricted.

18. Enter the name-dialing prefix. Test the scenario where there are multiple name matches. Make a valid selection and leave ThruDial.

19. Press the * key to retrieve a help prompt. Enter the name-dialing prefix and then press the * key to retrieve a different help prompt. Finally, press the # key to leave ThruDial.

20. Press the 0 key, and then the # key to leave ThruDial.

21. Enter the name-dialing prefix, and then press the 0 key to leave ThruDial.

**Other Types of Test Cases**

22. During the call session, do not key in any input and wait for the timeout sequence for rotary callers.

23. Create an application where some inputs are required in another voice-service component prior to entering ThruDial. When ThruDial is active, do not key in any input, and wait for the timeout sequence for touch-tone callers.

To streamline the testing process, we designed each of the MSCs to test as much functionality as possible. For instance, as shown in Figure 5.21, we created test case #1 to perform the following tasks:

1) verify the default greeting prompt for name dialing,
2) check the case where there is zero name match,
3) check the scenario where there are multiple name matches, and
4) exit via the Success output.

However, the ‘modular’ way of testing did not always apply, as some functionality of the system could be covered only by testing multiple components together. For instance,
the goal of test case #23 was to verify the initial timeout behavior for touch-tone callers when no keys were pressed. To test this behavior, we created a ServiceCreator application model that required a caller to enter some inputs in another component, such as PasswordCheck, prior to entering ThruDial. Figures 5.22 and 5.23 show the MSC created for test case #23 and the graphical view of the application, respectively. This MSC ensured that ThruDial executed the timeout sequence for touch-tone callers, instead of that for rotary callers. In our study, more than 20 test cases of this type were created. In fact, most of the time in the test case derivation phase was spent identifying relationships between the voice-service components and ensuring that all such relationships were considered. In general, behavior that involved accessing or manipulating the persistent data had to be tested using this type of test cases.
Figure 5.21: The MSC created for test case #1.
Figure 5.22: The graphical view of the application created for test case #23.

Figure 5.23: The MSC created for test case #23.
5.3.6 **Specification and Implementation Problems in ThruDial**

The SDL model and the experience gained from the formalization process allowed us to identify specification errors that escaped a series of manual inspections by the Nortel engineers. As the modeled components were not particularly complicated, most of the errors we found were caused by vagueness and missing information. In fact, the most time-consuming part of the formalization process was to understand the natural-language specification and to consult the engineers for clarifications. We estimated that these activities took as much time as the formalization process. In our study, we reported a specification error if

- a missing requirement affected the understanding of system behavior,
- a requirement conflicted with other parts of the specification, or
- a requirement that is presented in an unclear or confusing manner.

The problems that we found in the original specification of ThruDial (see Appendix B.2) during the formalization process are listed below; closely-related errors are grouped into one item.

**Specification Problems**

**General Errors**

1. The content and the length of various timeout prompts were not mentioned, and the document referring to the ThruDial specification did not contain this information.
2. When a caller pressed the * key, it was unclear whether ThruDial cleared the digit entries that she previously entered.
3. The specification incorrectly stated that a caller left via the output path 0 (*Attendant*), instead of *No Response*, when a timeout occurred.
4. The specification did not state how ThruDial handled the case in which the system transferred a caller to a valid but busy DN that did not have the “callforward busy” feature.
5. When ThruDial transferred a caller to a valid DN, she was “given an option to retry” if the transfer failed. It was unclear how the caller could retry and whether a help prompt was available at this state. The scenario where the caller stayed idle was also not explained.
**Fixed-length Number Dialing Errors**

6. The minimum fixed length should be one, instead of zero.

7. The number-padding algorithm was not explained clearly. For instance, if the administrator set the fixed length to be five and the padding string to be “123”, and the caller entered “987”, it was unclear whether the resulting string should be “12987” or “23987”.

8. Consider the case where the length of the padding string is the same as the fixed length. If the caller pressed # as the first key or stayed idle, it was unclear whether the system would transfer her using the padding string, since the fixed-length requirement was already met.

9. If the length of the padding string was greater than the fixed length, it was not clear how ThruDial performed the left-padding.

**Other Number Dialing Errors**

10. In the variable-length dialing mode, it was unclear how the system responded after the caller entered more than 15 digits.

11. After a caller keyed in the first digit, the action of pressing the * key added a pause to the existing DN entry. Thus, the help prompt could not be context sensitive, and its content shown in the specification was inappropriate. Moreover, it was unclear whether the pause entry was included in calculating the length of the DN.

**Name Dialing Errors**

12. When there were two users with identical names, it was unclear how a caller could locate the person that she wished to call.

13. When there were too many name matches (i.e., more than nine), the system played an error prompt. At this stage, it was unclear what actions the caller could take and what the corresponding system responses would be.

14. If a caller pressed the # key, and there were zero name matches, it was unclear how the system should respond.

15. If a caller pressed the # key, and seven persons were found, it was unclear how the system should respond if the caller pressed the # key, the * key, or an invalid digit (e.g., “8” or “9”).
16. The set of characters that could be used for creating user names was never mentioned. This information was important because a caller could search only alphanumeric characters.

"Name or Number" Dialing Errors

17. The restriction on the domain of the name-dialing prefix was never mentioned. For instance, the prefix cannot start with a zero as that would conflict with the request for an attendant.

18. It was unclear whether the name-dialing prefix was just a simple prefix or acted as a mode switcher that brought ThruDial from the number dialing model to the name dialing mode. For instance, if a caller entered the prefix and then pressed the * key, the specification did not mentioned whether the number dialing or the name-dialing help prompt would be played.

19. If a caller entered the 0 key after keying in the name-dialing prefix, it was unclear whether she should leave ThruDial via the 0 (Attendant) output—while the 0 key was the first entry of the DN, it was not the first digit key that she entered.

Using test cases derived from the formal model, we tested the implementation and reported behavior that did not agree with the specification. However, we intentionally did not report on the concepts that were missing or unclear in the specification but were rectified in the implementation (e.g., the left-padding algorithm in ThruDial), as designers had already realized the existence of these problems. Clearly, the testers would need to interact with the developers to clarify these problems, spending extra time in the test case execution phase. The errors that we found in the first version of the implementation are listed below; closely-related errors are also grouped into one item.

Implementation Problems

General Errors

1. The content of some of the help prompts, timeout prompts, and errors prompts did not agree with the specification.

2. The maximum number of timeouts before we left ThruDial via the No Response output was inconsistent with the specification.
3. An un-interruptible prompt "That number is invalid" was played when we were transferred to an invalid DN. This conflicted with the requirement claiming that all prompts were interruptible.

**Variable-length Number Dialing Errors**

4. We could enter a DN with more than 15 digits.
5. The * key, even when it was not used as the first key entry, did not act as a pause character. For instance, we were not transferred to phone number "8037" when we entered "8*037". Surprisingly, we found by trial and error that the input "8*xxx037" was needed, where xxx could be any three-digit number.

**Fixed-length Number Dialing Errors**

6. Even if we pressed the # key explicitly, ThruDial did not transfer us to a DN that was shorter than the fixed length.
7. We could enter a padding string with more than four digits.
8. If the length of the padding string was greater than the fixed length, the system immediately played the prompt "The call cannot be completed" and terminated the call session. From a caller’s perspective, we felt that this behavior was unreasonable.
9. Assuming that "8037" was a valid DN, and we set the fixed length to be 5 and the padding string to be "8037". If we entered "1" during the call session, the system transferred us to the DN "8037", even though the resulting DN should be "80371".

**Name Dialing Errors**

11. When we stayed idle, the system attempted to find a single match; it did not just play the timeout prompt.
12. It was never mentioned that the * key terminated the name entry when it was not used as the first entry. Instead, it acted like the # key.
13. Consider the case where a name search resulted in three entries and the system played the prompt "Three names have been found". If we stayed idle at this state, we got only one timeout before leaving ThruDial. However, if we pressed the * key and then stayed idle, we then got two timeouts, behaving inconsistently with the first scenario.
14. If we entered "0#" at the beginning of the call session, we did not leave ThruDial as expected. Instead the system played the prompt "0 names found; press * to see list; press # to select names."

15. The prompt "0 names found; press * to see list; press # to select names." was also played if a name search resulted in no match. Clearly, this prompt was a template for multiple name matches and did not apply to our situation (e.g., the last two sentences were inapplicable).

During the testing phase, a recurring pattern that we encountered was that some of the specification problems in ServiceCreator indeed propagated into the implementation. Using ThruDial as an example, the unclear/missing descriptions of the timeout prompts, the processing of the * key in number dialing, and the left-padding algorithm in fixed-length number dialing gave rises to numerous problems in the implementation.

The criticality of the problems in the specification and the implementation varied, ranging from incorrect content of the prompts to unknown behavior or even system crashes. For instance, in the ThruDial component, several problems that we found were about the unknown behavior of the name-selection state and the left-padding algorithm. In the Announcement component, we discovered that the rotary timeout never happened in one obscure scenario. Moreover, within the PasswordCheck component, we found that a malicious caller could crash a ServiceCreator application by entering an abnormally long password. Since we used the test cases derived from the SDL model to identify errors in the implementation after the Nortel engineers had officially completed testing, we were able to observe that these implementation errors, and many others, had not been discovered. The reason was that the Nortel test cases were derived from the same incomplete specification and missed these errors.

5.4 Findings

We began our analysis by seeking quantitative evidence to measure the effects of the formalization process on productivity. However, as the study progressed, we felt that it was also crucial to identify the qualitative factors (e.g., perceived usability of SDL and commitments from the development team) and the limitations of the study in order to reach accurate and unbiased conclusions.

Unfortunately, because of the exploratory nature of our study, Nortel engineers did not
keep track of many essential metrics, and we could not create a controlled environment where such metrics could be obtained, as mentioned in Section 5.2. In particular, we did not know the exact amount of time it took to fix a bug, if it was found during the inspection vs. design vs. testing phase. The lack of metrics impaired our ability to draw accurate quantitative conclusions.

5.4.1 Quantitative Results

The entire modeling process, which consisted of activities such as understanding and formalizing the specification as well as deriving and executing test cases, took about two person-months to complete. During this period, we kept track of a variety of productivity data in the study (column two of Table 5.2) for comparison with similar information from the existing development process (column three). Effort measurements in this table are approximated to the nearest person-day.

<table>
<thead>
<tr>
<th>Productivity Data</th>
<th>Study</th>
<th>Existing Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to model (person-days)</td>
<td>11</td>
<td>N/A</td>
</tr>
<tr>
<td>Time to inspect (person-days)</td>
<td>N/A</td>
<td>50</td>
</tr>
<tr>
<td>Number of specification errors reported</td>
<td>56</td>
<td>N/A</td>
</tr>
<tr>
<td>Number of test units</td>
<td>269</td>
<td>96</td>
</tr>
<tr>
<td>Time to create test units (person-days)</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Time to execute test units (person-days)</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>Number of implementation errors identified</td>
<td>50</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 5.2: Productivity comparisons.

Since the sizes of test cases varied greatly, we did not use a "test case" as the unit of comparison for testing-related data. Instead, we counted test units, the smallest externally visible functionality of the system, in each test case to ensure a fair comparison. Highlights from the table are summarized below:

- The time to model value included only the time used for modeling the SDL processes. While the modeling task did not have an exact equivalent in the actual development process, manual inspection was a similar activity that was also performed at the completion of the specification phase. Certainly, the formalization process was not meant to be a complete replacement. However, if a large number of specification errors were identified in a relatively short amount of time, the modeling task could be considered a way to decrease the time for inspection. (We discuss this point later in this section.)
The number of specification errors reported could not be used for comparison as the Nortel engineers did not keep track of such statistics.

More test units could be derived from the SDL model (which translated to better test coverage) in roughly the same amount of time, possibly because the model eased the creation task by providing a more in-depth understanding of the system as well as a better sense of completeness. One other reason for the difference in the quantity was that the test units from Nortel were sometimes vaguely specified (see Figure 5.24); the missing details contributed to a decrease in the number of test units reported.

Create an application that uses a ThruDial component with the “Allow dial by number or name” setting. Call the application and verify that the caller is able to spell the name of the party to be called or simply dial her DN.

Figure 5.24: A Nortel test case created for testing ThruDial.

The time needed for test unit execution in our study was much smaller for two major reasons: the derived test units were more detailed and thus easier to execute, and it was observed that Nortel engineers spent a lot of time revising the existing test cases because of the changes in requirements and creating more detailed test scenarios based on the vaguely specified test units. However, as illustrated in the interview study in Chapter 2, due to tight schedules, most testers did not document these extra units until the end of the entire testing phase, which spanned over more than four months. They admitted that some of these test units would inevitably be lost, contributing to a decrease in their total number.

The number of implementation errors identified in the study was two times larger than that of the existing development process. Many of them were missed because testers created test cases from an incorrect and incomplete specification, as indicated in the third row of Table 5.2. Problems such as incompleteness propagate to the test cases and affect the test coverage. In fact, 18 of the 50 problems could be linked to problems in the specification. Their criticality varied, but most of these errors resulted from undocumented assumptions or vaguely described requirements in the specification.

To obtain an accurate cost/benefit figure, we needed to collect additional statistics
such as an average cost to fix a requirements error discovered during the inspection and the implementation phases. As we mentioned earlier, the Nortel development team did not keep track of such metric; however, a conservative cost/benefit estimation was still possible. Without taking the improvement in software quality into account, we estimated the cost of formalization by subtracting the time of the modeling task from the direct savings in

- the test unit creation (0 person-days),
- the test unit execution (9 person-days), and
- the manual inspection.

The formalization process did not include a manual inspection phase, whereas the actual development took 50 person-days for it. The modeling task would come at no cost if it were to reduce the manual inspection by 2 person-days, or 4%. Because of the small necessary reduction, it was safe to assume that even in the short-term, the use of SDL could improve the quality of software without lengthening, and possibly shortening, the development cycle. Of course, the actual cost/benefit figure would be significantly more promising if the long-term benefits, coming from a better quality of the product, ease of maintenance and regression testing, and an ability to reuse a good specification, were taken into account.

5.4.2 Qualitative Observations

While all the quantitative data was in favor of the use of the formal modeling, it was clear that these results alone constituted only a part of our findings. Some observations from the formalization process that were not evident from Table 5.2 are discussed below.

- In addition to missing, inconsistent and ambiguous requirements, a recurring problem that we encountered in the ServiceCreator specification is the abundance of unrelated information. For instance, the ThruDial specification often diverts the focus from explaining the core functionality to providing irrelevant information, such as suggesting the types of voice-service components that should be connected to the output paths. However, the extra information complicates the specification and increases its length unnecessarily, while does not enhance the understanding of the requirements. From our experience in the formalization process, we believe that the use of formal requirements as a part of the specification helps the designers
concentrate on explaining only the essential knowledge, making the specification more concise.

- The most frequent complaint from the test engineers is that the missing information in the specifications often complicates the task of the test case creation. The SDL models encourage and assist developers in stating system requirements more precisely and completely, which should allow testers to create better quality (e.g., more detailed and with expected results more clearly defined) test cases and reduce the time needed for test case creation and execution. Developers should also benefit from the more complete specifications during the design and the implementation phases. This is an area where SDL can potentially improve the development process significantly. In fact, SDL has been successfully applied in the telecommunications field: from the traditional use of protocol specifications [BHS91, Sar89, FHH96] to high level specification [Man93], prototyping [VKK+98], design [MP98], code generation [Fro93], and testing [GM91] of telecommunications applications. Although the results reported in these studies were similar to ours, the goal of the studies was different: they were aimed to investigate the technical advantages and feasibility of SDL in a given environment, usually emphasizing on only one of the development activities.

- As with any other FDTs, a successful integration of SDL into the development process requires a firm commitment from the entire development team. For instance, the developers must ensure that the SDL model is always kept consistent with the system requirements and the code, e.g. last-minute changes in the design and implementation are propagated back to the model. Testers also need to ensure that their test cases always reflect the model accurately. While this is possibly one of the biggest obstacles in applying an FDT, the traceability provided by SDT should ease these tasks, and the advantages offered by the use of SDL should justify the extra effort.

- Compared to other FDTs, the strengths of SDL lie in its ease of use and the ability to express nontrivial behavior in a reviewable notation. For our study in particular, it can be used to express the requirements of reactive telecommunications systems effectively. Unlike many other FDTs, SDL does not require an explicit use of formal logic. The graphical user-friendly notation allows developers without a strong mathematical background to create FSMs easily. Compared to natural-language specifications, such FSMs give a much better sense of completeness, allowing to
detect missing scenarios (e.g., specification problem #5 in Section 5.3.6) with less effort. In addition, the formal syntax helps clarify ambiguities or inconsistencies (e.g., specification problem #11 in the same section). However, SDL tends to blur the line between requirements and designs. If proper abstractions are not applied, the model may become too detailed and unnecessarily large, possibly duplicating the design effort.

5.4.3 Limitations

Based on the opinions expressed by the Nortel developers and testers, ServiceCreator was representative of the types of systems they have to work with, so we were fairly confident that the results of the study would apply to similar projects in this environment. We were also fortunate to find a formal description method that provides adequate modeling and testing support and is well suited for modeling telecommunication systems. However, it would be difficult [ZW98] to generalize our findings outside the Nortel domain, since they would depend on the current development methodology, types of applications to be modeled, and the choice of a suitable formal description method.

Other limitations came from the fact that we had prior experience with SDL and were not constrained by development pressures. That is, we took the time necessary to produce high quality models and detailed test cases and felt that the process was straightforward. If time pressures prevent Nortel developers from applying the modeling techniques carefully, they may not achieve equally good results. In addition, novice users of SDL would take more time and possibly create less effective models of their systems, at least initially. However, appropriate training and availability of an on-site SDL expert during the transition phase can ensure a smooth modeling process.

5.5 Summary

In the last part of the formal specification pilot study, we used SDL to formalize the behavior of a multimedia-messaging system in a commercial setting. Although we did not have access to some development metrics to fully quantify our findings, the results of the study showed that the requirements of ServiceCreator was formalized effectively and economically, yielding improvements over the existing development process.

In particular, we were able to show that the use of SDL increased the quality of
the software without lengthening, and possibly shortening, the development cycle, even in the short-term. This was done by amortizing the cost of creating the model to the testing phase, i.e. deriving test cases from the formal model. On the other hand, the total decrease in the development cycle is only achievable if the formalization can be done fairly inexpensively: we formalized only selected components and stayed at a fairly high level of abstraction to keep the SDL model simple. Of course, the cost/benefit of the formalization would be more promising if the long-term benefits were taken into account.
Chapter 6

Usability Workshop

6.1 Overview

There is no doubt that the usability of formal description methods plays an important role in their acceptance in industry [Hei98]. An easy-to-use method encourages experiments and reduces the cost of integration. More importantly, the reality is that practitioners do not try to adapt to an inconvenient technique—they simply abandon it [WW93]. Thus, it is essential that SDL is perceived to be usable by Nortel engineers. Only then will they be willing to apply it to their projects.

To collect more information about the usability of SDL, we conducted a one-day workshop with six Nortel engineers and asked them to model two small systems in SDL and fill out a questionnaire. Since we intended to conduct more similar workshops before presenting the results in detail, this chapter contains only a very brief overview of this work-in-progress. The goals of the workshop and the demographics of the participants are discussed in Section 6.2. In Sections 6.3 and 6.4, we describe the two main parts of the workshop and provide a glimpse of the preliminary results. Section 6.5 summarizes this chapter.

6.2 Description

While the formal specification pilot study illustrated that some major problems in the Nortel development process could be tackled by using formal description methods, and that SDL could be applied to increase productivity, usability was an important issue that was not fully addressed. After the conclusion of the study, we conducted a one-day SDL
workshop with six Nortel engineers to explore

- Nortel engineers' initial views of the usability of SDL,
- the effectiveness of SDL in helping the Nortel engineers identify more specification errors and create test cases with better coverage,
- opinions about the applicability of SDL in solving problems in the Nortel development environment, and
- the perceived problems in integrating SDL into the development process.

Two software designers and four software testers volunteered to participate in the workshop. Table 6.1 shows a summary of their backgrounds.

<table>
<thead>
<tr>
<th></th>
<th>Subject1</th>
<th>Subject2</th>
<th>Subject3</th>
<th>Subject4</th>
<th>Subject5</th>
<th>Subject6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest Level of Education Achieved</td>
<td>Bachelor in Electrical Engineering</td>
<td>Bachelor in Electrical Engineering</td>
<td>Bachelor in Electrical Engineering</td>
<td>Bachelor in Computer Science</td>
<td>Diploma in Electronics Engineering</td>
<td>Masters in Information Systems</td>
</tr>
<tr>
<td>Years of Industrial Experience</td>
<td>8 years</td>
<td>3.5 years</td>
<td>7 years</td>
<td>1.5 years</td>
<td>13 years</td>
<td>3 years</td>
</tr>
<tr>
<td>Title</td>
<td>Software tester</td>
<td>Software engineer</td>
<td>Senior software engineer</td>
<td>Software tester</td>
<td>Software testing team leader</td>
<td>Software tester</td>
</tr>
<tr>
<td>Previous exposure to formal description methods</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Limited, from university courses</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 6.1: Characteristics of the participants.

6.3 Modeling Exercises

Since most of the subjects did not have prior experience with SDL, we began the workshop by presenting a four-hour overview of formal description methods and SDL. Afterwards, we provided the subjects with the natural language descriptions of two small software systems and asked them to model the systems using SDL. The first system, shown in Figure 6.1, was intended to be simple and acted as a warm-up exercise. Numerous missing requirements (or unstated assumptions) were seeded in the statement of this exercise intentionally.
A system provides a fax service to a caller who has a telset with two buttons: '1' and '2'. After detecting the presence of a caller, the system plays an initial announcement and requires her to specify the fax delivery method by pressing either the button '1' for same-call delivery or the button '2' for call-back delivery.

The system delivers the fax immediately over the telset if the first delivery method is chosen. For call-back delivery, the system obtains a three-digit number from the caller before proceeding to the actual delivery.

Figure 6.1: The natural-language description of the first software system.

Two GUI clients, client1 and client2, access a file in a remote server by issuing a combination of the three available commands: open, close, and store. The server reports the success or failure of each command by sending back an acknowledgment.

Besides acting as a remote storage, the server also guarantees the integrity of the file by disallowing invalid operations. For instance,

- a client cannot save a file without opening it first;
- if client1 has just saved a file, client2 is not allowed to overwrite it until it first reads the updated file.

For simplicity, assume that the server processes only one command at a time.

Figure 6.2: The natural-language description of the second software system.

The exercise was to be completed in teams of two. After inspecting the description manually, we asked the subjects to identify problems in the requirements and create test cases that could test the described system adequately. Then, the subjects were instructed to perform the same activities after modeling the same system in SDL. The exercise was setup in this configuration to investigate if the subjects could discover more requirements errors and create better test cases after using SDL. After finishing the first exercise, the subjects completed a similar exercise with a slightly more complicated software system (see Figure 6.2). Although we did not impose any time limit, the subjects were instructed to read and analyze the descriptions of the systems as if they were typical Nortel specifications, discouraging them from devoting excessive or inadequate attention to the exercises.

Analyzing the results from the exercises, we found that the subjects were able to discover additional specification errors and create test cases with better coverage after formalizing the behavior; some of them found even more errors than we originally seeded, i.e. the descriptions contained errors that we did not anticipate. A few minor usability
problems were noted, but the general consensus reached among the subjects was that the use of a formal, yet user-friendly, notation could help uncover problems hidden in the seemingly simple exercises, much more effectively than manual inspections.

6.4 Questionnaire

In the second part of the workshop, we asked the subjects to fill out a questionnaire that contained a mix of multiple-choice and short-answer questions. The main goal of the questionnaire was to obtain opinions from the subjects about the usability of SDL and its perceived role in the development environment. Table 6.2 lists some of the major results; its right column contains an average score on the scale from 1 (strongly disagree) to 5 (strongly agree).

<table>
<thead>
<tr>
<th>Statement</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL is easy to use</td>
<td>3.7</td>
</tr>
<tr>
<td>SDL can be used to address many of the development problems we are facing</td>
<td>3.7</td>
</tr>
<tr>
<td>The use of SDL increases our understanding of the requirements and their quality</td>
<td>4.2</td>
</tr>
<tr>
<td>The use of SDL lengthens the requirements analysis phase</td>
<td>4.0</td>
</tr>
<tr>
<td>The use of SDL shortens the design, implementation, and testing phases.</td>
<td>4.1</td>
</tr>
<tr>
<td>Integrating SDL into my work routine is worthwhile and should be tried.</td>
<td>3.7</td>
</tr>
</tbody>
</table>

Table 6.2: Some results from the questionnaire.

Preliminary results from the questionnaire strengthened our findings that SDL is a user-friendly formal description method that can be used effectively by Nortel engineers to improve their development process. Due of the small population size and also to reinforce our confidence in the results, we planned to conduct more similar workshops with other Nortel development teams before we analyzed the presented the data in detail. However, encouraged by the early results and the prospects of SDL, Nortel and the University of Toronto are already in the process of setting up another joint project to investigate the use of SDL further, explained further in Chapter 7.
6.5 Summary

Usability was one of the important issues that was not fully addressed in the formal specification pilot study. We conducted a one-day SDL workshop with the Nortel engineers to obtain their opinions about the usability of SDL, its effectiveness in assisting them to identify specification errors, and its applicability in their development environment. First, the subjects were asked to model two small software systems in SDL after inspecting them manually. We found that the subjects were able to identify additional specification errors and create test cases with better coverage after formalizing the system behavior. Then, they were asked to fill out a questionnaire designed to get their opinions about SDL and its perceived role in the development environment; preliminary results showed that SDL is a user-friendly formal description language that can be used effectively by Nortel engineers to improve their development process. We planned to conduct more similar workshops to collect more data to reinforce our confidence in the results.
Chapter 7

Summary and Future Work

In this thesis, we described a joint research project between the University of Toronto and Nortel for demonstrating the feasibility of formalizing requirements in a commercial setting. By conducting a case study, we tried to exemplify that FDTs can be used in a commercial environment by following some guiding principles to tailor their uses.

To ensure that FDTs were a suitable solution for concerns raised by a Nortel test manager about his development environment, we conducted interviews with Nortel engineers to identify problems with their process, ensuring that errors in the software specifications indeed constituted a major part of the problems before proceeding further. The results from the interviews allowed us to locate major areas for improvement, from which we derived criteria for conducting a board formal description method survey. After narrowing down the choices to three candidates, we applied these methods to model a small part of a Nortel multimedia-messaging system, called ServiceCreator, to familiarize ourselves with their strengths and weaknesses in detail. We, together with the Nortel engineers, chose ServiceCreator because it could be easily formalized and yet was representative of typical Nortel systems. Using our experience in modeling with the three methods, the feedback from Nortel engineers, and the criteria proposed for the survey, we applied a quantitative feature analysis to select a method that was usable in the Nortel development environment, was suitable for modeling telecommunications applications, and could effectively tackle the identified development problems.

We applied the chosen method, SDL and SDT, to formalize a part of ServiceCreator in parallel with the actual development process; test cases were derived from the resulting formal model. To quantify the benefits, we collected productivity data such as the time to create the model, the number of specification and implementation errors identified,
and the number of test cases derived to compare with similar data from the existing development process, provided by the Nortel engineers. Although we did not have access to some development metrics to fully quantify our findings, the results of the study clearly showed that software requirements were formalized effectively and economically. We also made qualitative observations about practical concerns in applying SDL and the advantages of SDL that were not apparent from the productivity data.

7.1 Lessons Learned

Agreeing with the proponents of FDTs [Hal90, BH95], results of our study reported an increase in software quality and productivity, in terms of the number of test cases generated, the time for creating and executing them, and the number of errors found in the requirements and implementation, even when SDL was applied in a commercial setting. Moreover, these benefits were obtained without lengthening and possibly by shortening the development cycle. The success of the project was in

- analyzing problems before proposing solutions—while this concept is seemingly intuitive, many FDT case studies tend to propose solutions without justifying the need or the appropriateness of those solutions. A careful analysis ensured that the we solved the correct problems using the best-known solutions.
- finding a suitable system—FDTs are not applicable to all types of systems. It was important for us to choose a system that had a good potential to benefit from the formalization process and yet was relevant to and representative of typical Nortel systems.
- using a well-chosen formal description method—it was important that we chose a method that could tackle the identified problems faced by the Nortel development team and was be able to formalize the requirements of the chosen system effectively. On the other hand, the method should fit into the Nortel environment as the Nortel engineers would be the eventual users.
- taking a lightweight formalization approach—we decreased the cost of formalization by utilizing a notation that was easy to use and review, formalizing only selected components, and staying at a fairly high level of abstraction. As ServiceCreator did not require a level of assurance necessary in safety-critical systems, we felt that it was unnecessary and ineffective to over-specify (e.g., specify the complete system)
and perform rigorous analysis (e.g., prove that system properties hold).

- leveraging the cost of formalization to the testing phase—FDTs inevitably increased the length of the requirements phase. By minimizing the cost of formalization and amortizing the cost of creating the model over the time-intensive software testing phase, the formalization process came virtually at no cost. The savings came from shortening the time for creating and executing test cases.

In our project, an important advantage of SDL was that the model could be used as a complement, not a replacement, of the existing natural language specification, allowing an evolutionary transition. This lets Nortel engineers conduct experiments with a small cost and without a huge commitment. However, as with any other formal description languages, a successful integration of SDL into the development process would require full co-operation from the entire development team and the ability of the engineers to apply the language effectively.

### 7.2 Limitations

Our project has a number of shortcomings:

- We did not integrate SDL into the actual development process to observe the impact of the formalization process on the entire development cycle. Nortel engineers were also not always involved in the modeling of ServiceCreator.

- The fact that we had a background in FDTs and were never under any development pressure might have positively affected the quality of the formal model and thus the derived test cases. In addition, our lack of prior knowledge in the telecommunications domain might have helped us detect subtle requirements errors better than the Nortel engineers [Ber95].

- We did not fully quantify our findings because the Nortel personnel did not allow us to create a controlled environment where we could collect some essential metrics.

- It was difficult to generalize the results from our project, since they would depend on the current development methodology, the types of systems to formalize, and the choice of a modeling method.

- The usability issue of SDL was not fully addressed by our formal specification case study, and we have not collected enough information from the usability workshop to evaluate the user-friendliness of SDL.
7.3 Improvements and Future Work

In the future, we plan to address some of the limitations of the project and expand the scope to focus on concerns that did not receive full attention in this thesis. By conducting more relevant studies that show both the benefits and the shortcomings of FDTs objectively, we hope to encourage more practitioners to experiment with FDTs.

- **Full Integration**

  Nortel and the University of Toronto plan to set up another joint project where SDL will be fully integrated into the development process. Starting from the requirement analysis phase, the Nortel engineers will apply SDL themselves, and we will only observe the progress and provide consulting, if necessary. This will let us examine the full effect of formalizing requirements on an evolutionary development cycle and to investigate how SDL affects other Nortel groups such as training and marketing.

- **Parallel Studies**

  We exemplified the benefits of FDTs by comparing the productivity data of our project with that of the existing development process. In fact, it is essential to have two control groups, so that the productivity comparisons can be made. Thus, if the Nortel engineers were to carry out the formalization themselves, another Nortel team would be required to work in parallel, using the existing development technique. This setting would have also been beneficial to our project, as it could have made our comparisons more direct and accurate.

- **Data Collection**

  The exploratory nature of our study prohibited us from having an adequate data collection process. In conducting future studies, we must ensure that a proper process is in place to collect productivity data that is essential for quantifying the benefits of FDTs, such as the number of requirements errors discovered in various development phases and the cost to fix a requirement error in the implementation phase. Usually, this must be arranged prior to the study as some development teams may not keep track of such statistics in enough detail.
• **Replication**

Since our current results are based on a single study, we need to conduct more case studies to investigate the use of SDL on telecommunications systems of different complexity to validate the results. To learn more about limitations and feasibility of FDTs in commercial setting in general, we should also experiment with applying FDTs to other application domains using the basic principles that proved to be valuable in our project, e.g., derive test cases from formal models and use FDTs in a lightweight manner. Clearly, we need to go through the entire process again as the goals of these studies may be slightly different, and we may have a different set of FDT selection criteria.

• **Usability**

The on-going series of workshops will allow us to determine if the Nortel engineers can use SDL effectively and whether SDL can indeed help them achieve better software quality. We should also conduct interviews or similar workshops with members of less technical Nortel groups to investigate the user-friendliness of SDL from their perspective, since they also need to access the software requirements specifications.

• **Verification**

Given the favorable results from our project, another interesting direction is to explore the use of rigorous techniques in verifying selected properties of complicated subsystems. Verification can encompass system-independent properties, such as freedom of deadlock or safety and security policies. Informal conversations with the engineers and the preliminary results from the questionnaire indicate that such techniques can be very helpful if they are applied in a cost-effective manner.
Bibliography


Appendix A

Interview Question Sets

This appendix shows the two question sets that were used for interviewing Nortel software designers and testers during the exploratory study. Readers can refer to Chapter 2 for more details about the study.

A.1 Question Set for Interviewing Software Designers

Background

Nortel and the University of Toronto have jointly proposed a research study in which the goal is to demonstrate the feasibility of applying a formal requirements specification technique in a commercial setting. The investigator, Andre Wong, is responsible for conducting and coordinating the study, while Nortel provides the necessary access to the engineering resources.

In this study, we want to investigate if the use of the new technique increases the quality of a requirement document and improves productivity in various stages of the software lifecycle. Three software developers and three software testers will be interviewed to ask about their general perceptions of problems in the Nortel development process as well as their knowledge of formal modeling languages.

The interview will be tape recorded.

Participants Rights

A summary of the collected results may be used for publication in research conferences and/or journals. We respect your rights as participants in academic research:

---

1Nortel, for the purpose of this interview, refers to the Toronto Multimedia Applications Center (TorMAC) of Nortel Networks.
the anonymity of your identity is strictly maintained. Under no circumstances will information that identifies an individual, group, or company be made available to anyone outside the research team without an explicit prior consent. Specifically, Nortel will only have access to results that are publicly available, and not to the raw data. However, it is possible that information that identifies individual subject can be obtained from the published data, because of the small number of subjects involved.

Although we hope that you answer all questions during the interview, you may skip any of them or withdraw from the interview at any time for any reason.

Education and Experience

I will start the interview by asking a few questions about your education and your work experience.

1. What is your educational background (e.g., B.Sc. in Computer Science from the University of Toronto)?

2. How long have you been working in the software industry (i.e., companies that develop/sell software for internal/external clients)? What is your title in Nortel? What are your responsibilities?

3. Describe two projects that you have worked on inside or outside of Nortel? What are your roles in these projects? What tools/programming languages have you used in these projects?

4. I am going to name a few phases of the software development cycle. First, pick three phases that you think are the most time-consuming. Then, rank those three phases on a scale from 1 to 3, where 1 represents the most time-consuming phase, and 3 represents the least time-consuming phase (relative to the other two phases).

   - understanding customer requirements,
   - translating customer requirements to system requirements,
   - writing the specifications,
   - performing specification reviews,
   - creating designs of systems,
   - creating implementations,
   - performing code reviews, and
   - debugging and testing systems.

5. Which of these phases do you enjoy the most? Which of these phases do you dislike the most?

Formal Modeling Languages

Definition: if we use modeling languages with formal syntax and semantics to reason about software, we can call them formal modeling languages. SDL, STATEMATE, Z,
VDM, and PVS are some of the more popular examples.

6. Are you familiar with any formal modeling languages? [If the answer is no, go to Question 9.]

7. I will read a list of formal modeling languages. Please stop me if you are familiar with some of the names.

<table>
<thead>
<tr>
<th></th>
<th>Never heard of it</th>
<th>Heard of it from where?</th>
<th>Received training from where?</th>
<th>Used it where?</th>
<th>In</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATEMATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PVS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Do you know any other formal modeling languages that are not in my list?

8. In general, how do you find formal modeling languages? Do you find them useful, difficult to use, or time consuming, etc? Please elaborate.

**Feature Specifications**

I will ask you a few questions about feature specifications.

9. Have you worked on an application called AppBuilder? Have you seen its specification? [If either answer is no, go to Question 11.]

10. Please give a brief description of the project that you are currently working on. [Call that project PROJ]

11. Please compare several aspects of the feature specification of PROJ to other feature specifications that you have seen. For the purpose of this interview, we will rate the quality based on the following criteria:
   - the specification contains many errors;
   - the requirements are explained completely and without ambiguity;
   - the design of the system/test cases can be easily derived from the specification.

Please rate the quality of the feature specification of PROJ with respect to the quality of other specifications that you have seen on a scale from 1 to 5, where 1 is poor, and 5 is excellent.

12. I will read a few statements about the feature specification of PROJ. Please tell me whether you agree with them using a scale from 1 to 5, where 1 is strongly disagree, and 5 is strongly agree.
• Quite often, the specification is too wordy. Information could have been presented more concisely.
• The specification is difficult to understand because the author often makes the wrong assumption about the level of understanding of the readers.
• Missing information is a major problem in the feature specification.
• Some requirements in the feature specification are often described vaguely or in inadequate detail.
• Some requirements are presented in an ambiguous manner (e.g., same word with different meanings in different parts of the specification).
• Inconsistency is a major problem in the specification (e.g., different parts of the specification conflict with each other).
• After reading the specification, it is often difficult to know, at a high level, what the system does.
• It is often difficult to derive test cases from the specification directly.

This is the end of my list. Can you think of other problems that you have encountered while reading the feature specifications, in general? You do not have to limit your response to the problems in the feature specification of PROJ.

13. I talked about potential problems in the specification. Do you have any suggestions for improving its quality?

14. Do you often find errors in the specification even after it is finalized?

15. In general, do developers revise the specification as soon as they find errors in it? [If the answer is yes, go to Question 16.]

   15.1 In what ways do you think the errors in the specification will impact the testers?
   15.2 What do you think will happen when the specification is used for building a new release of the same system?

16. Do you agree that most implementation errors originate from errors in the specification?

17. I will read a few statements about how a high quality specification can assist you in different development activities. Please tell me whether you agree with them on a scale from 1 to 5, where 1 is strongly disagree, and 5 is strongly agree. A high quality specification is helpful in

   • reducing the number of unnecessary interactions between the readers and the creators of the specification,
   • improving the design of the system (e.g., less unnecessary design changes),
   • improving the implementation (e.g., reduced number of errors),
   • shortening the white-box testing phase, and
   • developing a new release of the same system.
18. In the existing development process, a feature specification goes through two internal reviews and two external reviews before it is finalized. However, it appears that these reviews cannot detect all problems in the specification. I will read a few statements about the review process. Please tell me whether you agree with them on a scale from 1 to 5, where 1 is strongly disagree, and 5 is strongly agree.

- In general, I have confidence in the review process. Errors in the specification are infrequent or are usually minor.
- Since not much time is allocated for reviewing the specification, we cannot accomplish as much as we want.
- Manual inspections have limitations. It is difficult to remove all problems during the review meetings just by inspection.
- There is a lack of guidelines to control the quality of the review process. Developers review the specification in their own ways, which is a problem if they are inexperienced.
- Some people do not pay much attention to the review process. It is just a formality to them.

This is the end of my list. Do you know any other factors that may affect the effectiveness of the review process?

19. For the same problem (there are errors in the finalized specification), I will switch the focus from the review process to the process of creating the specification. After I read a few statements about that process, please tell me whether you agree with them on a scale from 1 to 5, where 1 is strongly disagree, and 5 is strongly agree.

- There is no incentive to remove the errors as most readers are already accustomed to the problems in the specification. Removing the errors does not make much difference to anyone.
- The quality of the specification is perceived to be good enough. There is not much room for improvement.
- A good quality specification is not considered to be important, as the quality of the design / implementation does not seem to be closely related to it.
- Since not much time is allocated for creating a good quality specification, we cannot accomplish as much as we want.
- The system is usually too complex for us to identify/remove all problems during the requirements phase.
- There is a lack of guidelines to control the quality of the specification. Developers usually write what they think are appropriate, which is a problem if they are inexperienced.

This is the end of my list. Can you think of any other reasons that may affect the quality of the finalized specifications?

We value your input. Do you have any general comments?
A.2 Question Set for Interviewing Software Testers

Background

Nortel and the University of Toronto have jointly proposed a research study in which the goal is to demonstrate the feasibility of applying a formal requirements specification technique in a commercial setting. The investigator, Andre Wong, is responsible for conducting and coordinating the study, while Nortel provides the necessary access to the engineering resources.

In this study, we want to investigate if the use of the new technique increases the quality of a requirement document and improves productivity in various stages of the software lifecycle. Three software developers and three software testers will be interviewed to ask about their general perceptions of problems in the Nortel development process as well as their knowledge of formal modeling languages.

The interview will be tape recorded.

Participants Rights

A summary of the collected results may be used for publication in research conferences and/or journals. We respect your rights as participants in academic research: the anonymity of your identity is strictly maintained. Under no circumstances will information that identifies an individual, group, or company be made available to anyone outside the research team without an explicit prior consent. Specifically, Nortel will only have access to results that are publicly available, and not to the raw data. However, it is possible that information that identifies individual subjects can be obtained from the published data, because of the small number of subjects involved.

Although we hope that you answer all questions during the interview, you may skip any of them or withdraw from the interview at any time for any reason.

Education and Experience

I will start the interview by asking a few questions about your education and your work experience.

1. What is your educational background (e.g., B.Sc. in Computer Science from the University of Toronto)?

2. How long have you been working in the software industry (i.e., companies that develop/sell software for internal/external clients)? What is your title in Nortel? What are your responsibilities?

3. Describe two projects that you have worked on inside or outside of Nortel? What are your roles in these projects? What tools/programming languages have you used in these projects?
4. I will name four phases of the test development cycle. Please rank them according to the amount of time you spend in each of them on a scale from 1 to 4, where 1 represents the most time-consuming phase, and 4 represents the least time-consuming phase.

- understanding the requirements specification,
- creating test cases,
- performing test case reviews, and
- executing the test cases.

5. Which of these phases do you enjoy the most? Which of these phases do you dislike the most?

**Formal Modeling Languages**

Definition: if we use modeling languages with formal syntax and semantics to reason about software, we can call them formal modeling languages. SDL, STATEMATE, Z, VDM, and PVS are some of the more popular examples.

6. Are you familiar with any formal modeling languages? [If the answer is no, go to Question 9.]

7. I will read a list of formal modeling languages. Please stop me if you are familiar with some of the names.

<table>
<thead>
<tr>
<th>Language</th>
<th>Never heard of it</th>
<th>Heard of it from where?</th>
<th>Received training from where?</th>
<th>Used it. in where?</th>
<th>In</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATEMATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PVS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Do you know any other formal modeling languages that are not in my list?

8. In general, how do you find formal modeling languages? Do you find them useful, difficult to use, or time consuming, etc? Please elaborate.

**Feature Specifications**

I will ask you a few questions about feature specifications.

9. Have you worked on an application called AppBuilder? Have you seen its specification? [If either answer is no, go to Question 11.]
10. Please give a brief description of the project that you are currently working on. [Call that project PROJ]

11. Please compare several aspects of the feature specification of PROJ to other feature specifications that you have seen. For the purpose of this interview, we will rate the quality based on the following criteria:

- the specification contains many errors;
- the requirements are explained completely and without ambiguity;
- the design of the system/test cases can be easily derived from the specification.

Please rate the quality of the feature specification of PROJ with respect to the quality of other specifications that you have seen on a scale from 1 to 5, where 1 is poor, and 5 is excellent.

12. I will read a few statements about the feature specification of PROJ. Please tell me whether you agree with them using a scale from 1 to 5, where 1 is strongly disagree, and 5 is strongly agree.

- Quite often, the specification is too wordy. Information could have been presented more concisely.
- The specification is difficult to understand because the author often makes the wrong assumption about the level of understanding of the readers.
- Missing information is a major problem in the feature specification.
- Some requirements in the feature specification are often described vaguely or in inadequate detail.
- Some requirements are presented in an ambiguous manner (e.g., same word with different meanings in different parts of the specification).
- Inconsistency is a major problem in the specification (e.g., different parts of the specification conflict with each other).
- After reading the specification, it is often difficult to know, at a high level, what the system does.
- It is often difficult to derive test cases from the specification directly.

This is the end of my list. Can you think of other problems that you have encountered while reading the feature specifications, in general? You do not have to limit your response to the problems in the feature specification of PROJ.

13. I talked about potential problems in the specification. Do you have any suggestions for improving its quality?

14. What do you do when you find problems (e.g., confusions, incorrect or missing information) in the specification?

15. Do you think that high quality specifications can help you in creating test cases? Please elaborate.
16. Do you think that high quality specifications can help you in executing test cases? Please elaborate.

17. Do you agree that most implementation errors originate from errors in the specifications?

The following questions are about creating test cases. Assume that you have just read a specification and are ready to create test cases. Please answer the questions according to your experience in the testing team.

18. What kind of test coverage do you try to achieve while creating test cases?

19. Assume that you create a set of test cases to test a given functionality in a software system exhaustively. Are there ways to verify that the test cases that you have created indeed cover all the scenarios adequately?

20. I will read a few statements about creating test cases. Please tell me whether you agree with them on a scale from 1 to 5, where 1 means strongly disagree, and 5 means strongly agree.

20.1 I create fewer test cases when time is tight. [If the answer is either 1 or 2, proceed to Question 20.2]

20.1.1 How do you determine which test cases to leave out?

20.2 I create vague test cases when time is tight. For example, instead of describing each test step of the test cases in detail, only high-level descriptions are given. [If the answer is either 1 or 2, proceed to Question 20.3]

20.2.1 If testers create vague test cases, do you think that the test case execution phase will actually be lengthened?

20.2.2 Do you think that the entire testing phase will also be lengthened?

20.3 I create fewer test cases if the component being tested is relatively unimportant.

20.4 I create vague test cases if the component being tested is relatively unimportant.

This is the end of my list. Can you think of any other factors that may affect the number of test cases created and their level of detail?

21. Indeed, it appears that some of the AppBuilder test cases are only vaguely specified. Excluding time as a factor, why do you think that testers create vague test cases?

22. During regression testing, how do you identify the set of test cases that need to be re-tested?

23. When there are changes in the requirements, do the developers notify the testers to update their test cases? How often do the testers update the corresponding test cases?
24. In the existing development process, test cases are reviewed before they are finalized. However, in some cases, it appears that these reviews cannot detect all problems in the test cases. For instance, some of the reviewed test cases are still extremely vague, or they try to verify obsolete requirements.

25. I will read a few statements about the review process. Please tell me whether you agree with them on a scale from 1 to 5, where 1 is strongly disagree, and 5 is strongly agree.

- In general, I have confidence in the review process. Errors in the test cases are infrequent or are usually minor.
- Since not much time is allocated for reviewing the test cases, we cannot accomplish as much as we want.
- Manual inspections have limitations. It is difficult to remove all problems during the review meetings just by inspection.
- There is a lack of guidelines to control the quality of the review process. Testers review test cases in their own ways, which is a problem if they are inexperienced.
- Some people do not pay much attention to the review process. It is just a formality to them.

This is the end of my list. Do you know any other factors that may affect the effectiveness of the review process?

26. For the same problem (there are errors in the finalized test cases), I will switch the focus from the review process to the process of creating test cases. After I read a few statements, please tell me whether you agree with them on the same rating scale as before.

- It is too troublesome and unnecessary to produce detailed test cases during the test case creation phase; testers can probably figure out the details during the actual testing.
- It is sometimes impossible to avoid creating vague test cases because of problems in the specification.
- There is often not enough time allocated for creating test cases. This results in lower quality test cases.
- There is a lack of guidelines to control the quality of the test cases. Testers usually create what they think are appropriate, which is a problem if they are inexperienced.

This is the end of my list. Can you think of any other reasons that may affect the quality of the finalized test cases?

We value your input. Do you have any general comments?
Appendix B

ServiceCreator Components: Original Specifications

This appendix shows the original specifications of the PasswordCheck and ThruDial components of ServiceCreator.

Disclaimer: The content of the specifications remains intact, except that references to Nortel products are removed and the figures are modified to comply with the non-disclosure agreement. In other words, the quoted text retains a lot of conceptual (e.g., inconsistencies and ambiguities) and linguistic (e.g., grammatical and spelling mistakes) errors from the original documents. Readers can consult Sections 3.3.1 and 3.3.2 for the high-level overview of these components.

B.1 PasswordCheck

Functional Overview

The PasswordCheck block controls access to certain parts of an application by prompting callers to enter the password before being allowed to continue. Like all other blocks, the administrator places this block anywhere in an application. In defining the block, the administrator enters the password that callers must enter. The administrator is then responsible for remembering the password and distributing it to those people requiring it.

The PasswordCheck block has the following outputs:
Figure B.1: Workspace view of PasswordCheck block.

- **Valid Password (1 through 5)**
  - this output indicates that the caller entered one of the valid passwords specified in the block's parameters, and exits on the respective output.

- **Max Invalid**
  - this output indicates that the caller entered the maximum number of invalid passwords
  - the caller should not be permitted to proceed

- **Max Invalid Session**
  - The caller has reached the maximum number of invalid password attempts allowed per session as set in the session profile.

- **No response**
  - the caller has made no response before the timeout.

- **# (Cancel)**
  - this output indicates that the caller cancelled the password entry by pressing the number-sign key (#) without pressing any digits
  - the caller should not be permitted to proceed

**Administrator perspective**

To setup the PasswordCheck block, the administrator must complete these steps:

1. The prompt played to the caller on entering the block must be defined and selected.
2. At least one of the application passwords must be selected. To select a password, the checkbox to the left of the desired password must be checked.
3. The password must be entered for all passwords that are enabled, and the respective outputs must be connected. To enter a password the administrator first types the
password (only numbers 0-9 are allowed). The administrator must enter the password again in the confirmation text edit box to ensure that the password was not mistyped. The numbers in the password will be echoed to the display as the asterisk character (*) to ensure proper security. The password will be encrypted prior to storage on the disk.

**Caller perspective**

**Immediate Prompt**

On arrival at the PasswordCheck block, the caller may hear a custom recorded prompt or the system default prompt. The greeting is intended to orient the caller and provide information on how to enter an extension number or name. If the system default prompt is chosen, the system plays: “Password:?"

**Delayed Prompts**

The caller may experience one of two types of treatment if no key is pressed when the caller is in the PasswordCheck block. If the caller has arrived at the PasswordCheck block without yet pressing any keys in the call, the no response condition is treated as a
A strong indication that the caller is not calling from a touch-tone phone. In order to avoid annoying rotary-dial callers, the PasswordCheck block exits after the first Command Entry timeout period. The application jumps to the Rotary Dial block and follows its output to the next block. Thus the administrator decides the handling for these “Rotary Dial Timeout” cases. Typically the call would be transferred to an attendant.

In the second case, where the PasswordCheck block is reached after keys have already been pressed during the call (e.g. the PasswordCheck block is called from a Menu), the block follows a more extensive timeout sequence. After the first Command Entry timeout period passes, a delayed prompt is played (e.g. “Please enter the password followed by number sign”). A second Command Entry timeout is followed by a second delayed prompt (e.g. “For help, press star”). Finally, a Short Disconnect timeout period follows. When this time lapses, the PasswordCheck block exits on the No Response output without playing any prompt.

Delayed prompts are context sensitive. If the caller has begun to enter the password, the delayed prompts will urge the caller to continue:

“If you have finished entering the number, press number-sign. For help press star.”

If nothing has yet been entered, the caller is encouraged to star:

“Please enter the password followed by number-sign.”

Help

Pressing star in the PasswordCheck block plays the help prompt:

“You have selected a service which requires a password. Please enter the password, followed by number-sign.”

“To cancel, just press number-sign.”

This same prompt is played regardless of whether part of the password has already been entered or not. If part of the password has been entered when the star key is pressed, it has the effect of clearing the entry thus far.

Cancel

If the caller presses the number-sign (#) key without entering any data (i.e. entering a null password), the block interprets this as a request to cancel out of the PasswordCheck block and the block exits on the Cancel output. The caller would typically be returned to a previous Menu from which the service with the password was selected.
Max Invalid Password Entries Per Session

As well as having a maximum number of invalid password entry attempts on each PasswordCheck block, there is an overall limit on the number of such attempts that can be made during a single call. This is to prevent a hacker from repeatedly trying to break through a PasswordCheck block.

If not for this session limit, callers might be allowed a limitless number of attempts to hack a password, because the Cancel output could be used to exit the PasswordCheck block before the block's limit is reached. Re-entering the block, re-starts the invalid count (for the block). When this limit is reached an event is generated and will form part of a Reporter alert or report 1. The Max Invalid Password Entries Per Session field is set in the Session Profile. The range is 1 to 99. The default is 10.

B.2 ThruDial

Functional Overview

The ThruDial block is used to create an automated-attendant service that transfers callers to the extension of their choice. It plays an optional greeting (either custom or default), collects an extension number or name, and then transfers the caller to that extension. Restriction/permission lists determine what type of extension number (internal and/or external) callers can enter.

![ThruDial Block Diagram]

Figure B.3: Workspace view of ThruDial block.

Multiple instances of ThruDial can be defined, each answering a different phone number or being invoked from a Menu block. This is useful in the following cases:

1) Users can provide different greetings on different lines.
2) certain dialing prefixes can be disabled on some lines and allowed on others, by using different restriction/permission lists. For example, one line may disallow all calls besides internal extensions while another may allow tie-line and local trunk access.
The ThruDial block supports three dialing methods: number, name, or either. With name-dialing, the user spells the person's name using the telephone keypad.

Extension numbers can be fixed or variable in length. Fixed length numbers can, optionally, be left-padded with a dialing prefix to make calls to other sites easier to dial.

Callers can dial both internal and external extension numbers (subject to user configured restriction/permission codes).

The ThruDial block has the following outputs:

- **Success**
  - this output indicates that the caller was transferred successfully. Since the call has been transferred elsewhere, there is no choice here but to end the session.

- **0 (Attendant)**
  - this output indicates that the caller requested an attendant by pressing the zero (0) key on the telset keypad. The next block would typically be a Transfer Call block. It is named "0 (Attendant)" to encourage administrators to connect it to something appropriate.

- **# (Cancel)**
— this output indicates that the caller cancelled the ThruDial operation by pressing the number-sign key (#) on the telset keypad. This typically is used to give the caller an option to return to a previous Menu block.

- No Response

— this provides call handling in the case of a caller not specifying any address (name or number). On the Command Entry Timeout (defined in the Application Builder profile), the call will proceed on this output.

**Administrator perspective**

To setup the ThruDial block, the administrator must complete these steps:

1) To specify the voice item to be used as the ThruDial greeting, the administrator clicks the left-hand side box under ‘ThruDial greeting’, and then types the name of the desired voice item (or select it from the drop down list). An entry in the drop down list is available for this field to allow the administrator to specify that no greeting prompt is required. Also available from the drop down list is the default system prompt. The ThruDial greeting is always interruptible. If a non-interruptible greeting is required, an Announcement (with key assignments set to “Ignore all keys”) can directly precede the ThruDial block. In this case, the No Greeting option should be selected.

2) If the administrator wishes to allow the caller to dial the number using the numeric keys on the telset keypad, they click the ‘Allow dial by number’ radio button. Otherwise, they go to Step 6. The ‘Use fixed length numbers’, ‘Length’ and ‘Left pad’ controls will be disabled unless this radio button is selected.

3) If the administrator wishes to force callers to enter numbers of a particular length, they must ensure that the ‘Use fixed length numbers’ checkbox is checked. If the “Fixed length number only” checkbox is checked, numbers will be dialled as soon as the specified number of digits are entered. With fixed length dialling shorter numbers can still be entered, but they must be terminated with a number-sign. If the box is unchecked, callers can enter numbers of any length (up to a maximum of 15 digits) but must terminate their entry with a number-sign. Otherwise they go to Step 6. If this checkbox is not checked, the ‘Length’ and ‘Left pad’ controls will be disabled.
4) To set the numeric length for this ThruDial, the administrator clicks the ‘Length’ box and types to specify the desired length. Numbers in the range 0-13 are accepted.

5) To enter the Left pad that will be applied to number entered by the caller, the administrator clicks the ‘Left pad’ box and type the left pad digits. For fixed length numbers, specifies an optional string of digits that are padded to the left of the number entered. Only as many digits as are needed to make up the specified fixed length are taken and placed at the front of the number. This field is optional, and accepts up to four digits (no other characters).

6) If the administrator wishes to allow the caller to perform the ThruDial by entering a name, they click the ‘Allow dial by name’ radio button. If “Allow dial by name” dialing is selected, callers must spell out the name of the person they wish to reach using the telephone keypad. Otherwise, they go to Step 7.

7) If the administrator wishes to allow the caller to specify either a number or a person’s name to ThruDial to, they click the ‘Allow dial by number or name’ radio button. If this option is chosen, the caller will be required to enter the name dialing prefix (default 11) prior to spelling the name. The name dialing prefix is configured in the Messaging Administration.

8) The administrator must select a restriction/permission list to apply to each ThruDial service. They select from a combo-box listing the available restriction/permission lists set up in the SMP for the customer. The administrator can select any list present, but cannot enter any new list name. DNs entered are compared against the restriction/permission list to determine if the DN is legal for this ThruDial block. The restriction/permission Lists are configured in the OA&M application.

Caller perspective

Greetings

On arrival at the ThruDial block, the caller may hear a custom recorded greeting or the system default greeting (no greeting is also possible). The greeting is intended to orient the caller and provide information on how to enter an extension number or name.

The system default greeting varies depending on whether the service is configured to accept a number, name or both, and whether the number is of fixed or variable length.

---

1Name dialing could possibly be selected, but not enabled on the system. In this case the system wide setting takes priority, and the ThruDial will not work, this is however an unlikely scenario.
If dialing is by number only and the DN entered must be of fixed length, the system prompt is:

"Please enter the extension number of the person you would like to reach."

If dialing is by number only and the DN can be of variable length, the system prompt is:

"Please enter the extension number of the person you would like to reach, followed by number sign."

If dialing is by name only, the system prompts:

"Please enter the name of the person you wish to reach, followed by number sign. To enter a name, spell the last name and then spell the first name."

Finally, if dialing is by number or name, the system prompt is:

"Please enter the number or the name or the person you wish to reach, followed by number sign. To enter a name, press (name dialing prefix), spell the last name and then spell the first name."

The system greetings are played in the current language of the application. That language is set using the Set Language block.

Name-Dialing\(^2\)

If name-dialing is invoked, the caller must enter the person's name by spelling out the last name first, followed by the first name using the numeric keys on the telephone keypad. As soon as a match is made (this will likely occur before the caller has finished entering the name), the system plays the message

"Calling (personal verification)"

and then places the call. If no personal verification is recorded, the system plays the letters of the name it is trying to reach, starting with the last name.

If the caller terminates name entry by pressing number-sign and more than one person has been found, the system prompts the caller to enter more of the name if they know it. If the caller indicates (by pressing number-sign) that they either have spelled the whole name or don't know any more of it, the system reports:

"(number) names have been found."

Or if the number of matches exceeds more than nine entries\(^3\), the system plays:

---

\(^2\)Name dialling is only available in countries with a Latin alphabet.

\(^3\)The OA&M uses a hard coded value of nine for the maximum number of name addressing/dialing matches to present when the user presses # after partial entry of a name. There was some discussions as to whether this would be an OA&M configurable field, but it is not included as OA&M release 1.0.
“Too many names have been found.”

In the former case, where the number of names found is not excessive, the system continues with a list of the names found:

“For (personal verification of person 1) press 1, for (personal verification of person 2) press 2, ... for (personal verification of person 5) press 5.”

Timeout On Name-Dialing  Name-dialing in the ThruDial block follows the “Name Dialing Timeout” sequence described in the “OA&M Release 1 Integrated Voice/Fax Messaging (MMUI) User Interface Feature Specification”. This timeout sequence plays a number of delayed prompts and then transfers to an attendant.

Help On Name-Dialing  Help can be obtained at any time while entering a name by pressing the star key. The help prompt is context sensitive. If the caller has not yet started to enter the name, the prompt is:

“You are about to enter the name of the person you wish to call. Start spelling the last name and then spell the first name. For Q press 7, for Zee press 9. To exit, press number-sign.”

If the caller has begun spelling the name, the help prompt is:

“You are spelling the name of the person you wish to call. If you have finished spelling the last name and the first name, press number-sign. For Q press 7, for Zee press 9. When the name is recognized, it will be announced.”

Number Dialing

Each ThruDial block can be configured to accept either fixed length or variable length extension numbers. If fixed length dialing is used, the caller does not need to terminate the number with a number-sign. Once the required number of digits are entered, the call is immediately made. Any extraneous keys (e.g. number-sign, entered after a number of the fixed length is received) are ignored. If variable length dialing is used, the system accepts any digit string terminated with number-sign as the number to dial. Numbers dialed can be either internal or external. External numbers have to be dialed with the appropriate prefix to reach external trunks. No dialing translations are done. The star
key can be used in a number (except as the first digit). When used in a number, it is treated as a pause character.

**Left Pad Characters** Dialing numbers in frequently called locations (e.g. a branch office) can be made simpler by using a ThruDial block. Normally, dialing other sites requires a long DN because the extension is prefixed by the digits that identify the location. Using the ThruDial block with fixed length number dialing and left pad characters provides a solution to this problem.

The left pad characters defined for the ThruDial block are added to the front of the number that is entered by the caller (i.e. they form a prefix). Thus, for calls to a branch office, a DN may be set up with a ThruDial block that left pads the routing digits (e.g. 6-123), so that callers need only enter the extension number.

Left pad characters work with fixed length number-dialing in such a manner that the number entered is padded with the left pad digits to bring the total length of the number up to the length required. Thus, if the DNs to be called from the ThruDial service are expected to be 8 digits (e.g. 6-123-4567) and the left pad contains 4 digits (e.g. 6-123), the caller need only enter the last 4 digits of the number terminated with a number-sign (e.g. 4567#). If the caller enters less than 4 digits, the system plays an error prompt:

"((Beep)) That number cannot be reached from this service."

Callers who are used to entering all 8 digits could continue to do so and would not need to enter a number-sign. In such a case, no digits would be padded.

**Timeout On Number Dialing** Number-dialing in the ThruDial block follows the "Modified Short Timeout" sequence described in the "OA&M Release 1 Integrated Voice/Fax Messaging (MMUI) User Interface Feature Specification". This timeout sequence plays a number of delayed prompts and then transfers to an attendant.

When number dialing is used with fixed length numbers and left-pad characters, the timeout sequence includes an extra step. When the caller pauses and hits the first timeout, the system checks to see if the number entered so far, when padded with the left-pad characters, constitutes a complete extension number (i.e. does it reach the specified fixed length?). If the number is long enough, the system assumes the caller has finished and tries to dial it. If the number is still too short, the system carries on with the regular delayed prompts.
Help on Number Dialing  Help in number dialing is activated when the caller enters star as the first digit (a star in any other position is interpreted as a pause character). When help is activated, the system help prompt plays information about how to enter an extension number.

If star is pressed after part of a number has been entered, the system plays:

"You are entering a phone number. If you have finished, press number sign. To reach an attendant, press 0."

Null Entry

If instead of entering digits for a number or to spell a name the caller enters just number-sign (i.e. a null entry), the ThruDial block interprets this as a request to cancel the selection. Thus, the ThruDial block exits on the # (Cancel) output, which is typically connected to a previous Menu.

Attendant Request

The attendant is requested in one of the following ways:

i. by entering a 0 in number-dialing and waiting for the timeout

ii. by entering 0# in number-dialing

iii. by entering 0 as the first character in name-dialing

When the attendant is requested, the ThruDial block exits on the Attendant output. It is left to the administrator to connect the Attendant output appropriately. Typically this would be to a Transfer Call block, which would transfer the call to an attendant.

Call Processing

Once the number has been entered, the following situations can occur:

Valid number, idle DN: If the number is valid, then the caller hears ringing for the dialed number. The ThruDial session is disconnected at this time. Note: if the phone set has “Call Forward No Answer” to voice messaging, then the caller reaches call answering (i.e. the transfer is complete and the ThruDial service is done).

Valid number, busy DN: If the called DN is busy, then the system plays the message “(beep) That number is busy. Please try again later.”
If the phone set has “Call Forward Busy” to voice messaging, then the caller reaches call answering (i.e. the call is treated as per Case 1).

Valid number, transfer fails: If the number is valid but the transfer fails (e.g. due to AML link failure), an error message is played and the caller is given the option to retry. They could at this point cancel out of the block, and exit on the cancel output.

Invalid number, vacant DN, mailbox exists (i.e. Guest Mailbox): If the number is invalid (not in service) but a mailbox exists for the number, then the caller is transferred to an express messaging session for this mailbox.

Invalid number, vacant DN, no mailbox: If the number is invalid (i.e. not in service) and no mailbox exists for the number, then an error message is played:

“\(\text{(Beep)}\) That number is not in service. Please try again.”

and the caller is prompted for another DN (i.e. the standard greeting is played).

Invalid number, restricted DN: If the number dialed is restricted (e.g. the DN is used by SL-1 features), then the system plays an error message:

“\(\text{(Beep)}\) That number cannot be reached from this service. Please try again.”

and the caller is prompted for another DN.

Invalid number, restricted dialing prefix: If the number dialed is one of, or begins with one of, the prefixes defined as “restricted” in the restriction/ permission list for the block, then an error message is played:

“\(\text{(Beep)}\) That number cannot be reached from this service. Please try again.”

and the caller is prompted for another DN (i.e. the standard greeting is played).
Appendix C

SDL: A Brief Description

This appendix provides readers with the necessary knowledge to understand the Specification and Description Language (SDL) [CCI88] models in this thesis by presenting a brief description of the language. Readers are encouraged to consult [EHS97, Ols94, CCI88] for detailed information on the syntax and semantics.

C.1 Background

SDL is a finite-state machine (FSM)-based formal description language that is suitable for specifying reactive systems in which operations can be best described by a series of discrete state changes. It was first standardized by International Telecommunications Union (ITU) in 1976 as Recommendation Z.100 and have received several updates since then (as recent as 1996). This standard defines the formal syntax and semantics but does not dictate the use of any particular methodology, although the appendix of the standard includes a recommended methodology guide [CCI92]. Models can be created in a textual form or using a flowchart-like graphical notation; only the latter is discussed in this appendix.

C.2 Overview

SDL models have three major layers: the system, block and process layers. Each SDL system contains interconnected blocks, which may be sub-divided into more blocks; each block may contain multiple processes. Figure C.1 shows the skeleton of an SDL model. These layers are used for organizing large systems hierarchically.
SDL processes are extended FSMs that execute concurrently with equal priority. They communicate with each other and the environment, i.e. systems outside the SDL model, asynchronously by passing signals through channels; only point-to-point communication, i.e. one sender and one receiver, is supported. Because of the hierarchical structure of the models, channels may need to pass through several layers before reaching their final destination, which can either be the environment or other processes. The arrows in Figure C.1 show the possible placement of the channels. In the next section, we use the SDL model of the system described in Figure C.2 to guide our brief introduction to SDL. This system ensures that the 10 line cards that it administers are used by at most one requester at any time.

The resource manager accepts requests to access a particular line card out of the 10 line cards that it administers. The manager denies requests to occupied line cards until they are released and ensures that only the requested line cards are released. In response to each incoming command, the manager sends an acknowledgment to notify the originator about the result of the command.
C.2.1 Systems, Blocks, and Processes

Figure C.3 shows the SDL system created for the resource manager (the italicized phrases and the dotted arrows do not belong to the SDL system). The name of the SDL system, *resourceManager*, is shown in the top-left corner of the Figure. The resource manager receives two types of commands, request and release, and generates two types of responses, grant and deny. These inputs and outputs are mapped to SDL signals, defined inside the declaration symbol using the keyword *signal* (see Table C.1 for a list of SDL symbols). Each of the input signals carries a parameter of the user-defined type *intRange*; this parameter denotes the particular line card that the requester wants to obtain or release. The keyword *syntype* defines this data type, which has an inclusive integer range of 1 to 10.

```
system resourceManager

data type declaration

data type

signal declarations

channel name

channel

name of the SDL system

signal request(intRange), release(intRange), grant, deny;

syntype intRange = integer

endsyntype;

requestChannel

[request, release]

accessControl

[grant, deny]

respondChannel

SDL block

name of the SDL block

signals that pass through the channel

Figure C.3: The SDL system of the resource manager.
```

This SDL system contains one SDL block *accessControl*, shown as a rectangle in the middle of the diagram. This block receives signals from the environment via the channel *requestChannel*, and the channel *respondChannel* carries signals from the block back to the environment. These channels behave like unbounded FIFO queues. Also, signals that need to pass through a particular channel must be explicitly specified in square brackets associated with that channel. For instance, *request* and *release* are the signals that can pass through the channel *requestChannel*.

Figure C.4 shows the content of the block *accessControl*. This diagram is similar to the system diagram except that an SDL process *accessControlProc* (shown as an octagon) is defined, and the channels need extra identifiers to associate them with the channels in the system layer. For instance, the ‘tail’ end of the incoming channel *inputChannel*
has a label requestChannel, implying that this end is connected to the 'head' end of
the channel requestChannel at the system layer.

Figure C.4: The SDL block of the resource manager.

We construct the SDL process accessControlProc using some of the symbols in Ta-
ble C.1, and its content is shown in Figure C.5. This table lists the symbols in column
1, their names in column 2, and their descriptions in column 3. The symbols stop, create
request, procedure call, procedure start, and procedure return are not used in this Figure.

Inside the declaration symbol of Figure C.5, the keyword newtype defines an integer
set data type; the keyword dcl instantiates a variable of this data type named LCSet for
representing the set of requested line cards and an integer variable LCNo for representing
the line card that a requester is interested in. The latter variable can only hold integers
within the range defined by intRange, and SDL tools check this invariant during the
run-time execution of the model. The SDL process begins at the start symbol and waits
for the incoming signal at the state idle. Upon receiving the signal request, the process
transfers the value inside the signal parameter to LCNo. Then, depending on the results
of the set membership check specified in the decision symbol that follows, one of the
outgoing paths is taken. If the 'false' path is taken, the set inclusion operation specified
in the task symbol is performed, and the process sends the signal grant. In the other
case, the signal deny is sent. These signals pass through channels outputChannel and
respondChannel before they are transferred to the environment. In both cases, the flow
of control goes back to the state idle, using the state symbol with the identical name in
the first scenario and the connector symbol named restart in the other. The situation for
receiving the signal release is similar to that of receiving the signal request. The dotted
open-ended rectangles near the middle of the diagram are the symbols for storing textual
A BRIEF DESCRIPTION

Figure C.5: The SDL process for the resource manager.

C.2.2 Other Features

SDL procedures, similar to those found in conventional programming languages, can be defined at the block layer or inside an SDL process. Their appearance are similar to those of the SDL processes, with the exception that different symbols are used to signify the starting and the ending points. The last three symbols in Table C.1 are procedure-specific.

A variety of data types is supported in SDL, from conventional types such as integer, string, and real to more complex types such as array, set, bag, and a C-like record type called struct. The declaration symbol on the left side of Figure C.6 defines the data types for an integer-indexed boolean array, an integer set, an integer bag, and a struct that has a boolean variable and a real variable as its data members; the declaration symbol on the right side instantiates variables of these data types.
**Table C.1: Some SDL symbols.**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>○</td>
<td>start</td>
<td>denotes the starting point of an SDL process</td>
</tr>
<tr>
<td>✗</td>
<td>stop</td>
<td>terminates an SDL process</td>
</tr>
<tr>
<td>○</td>
<td>state</td>
<td>represents a state; only the input symbol can follow it</td>
</tr>
<tr>
<td>□</td>
<td>input</td>
<td>waits for a signal; it can only be used right after a state symbol</td>
</tr>
<tr>
<td>□</td>
<td>output</td>
<td>generates a signal</td>
</tr>
<tr>
<td>□</td>
<td>task</td>
<td>performs an operation such as a variable assignment</td>
</tr>
<tr>
<td>□</td>
<td>decision</td>
<td>branches to different paths according to the condition specified in the symbol</td>
</tr>
<tr>
<td>○</td>
<td>connector</td>
<td>links two symbols together; it must be used in pairs, and the destination end cannot be an input symbol</td>
</tr>
<tr>
<td>□</td>
<td>declaration</td>
<td>declares constructs such as data types, constants, variables, and signals</td>
</tr>
<tr>
<td></td>
<td>comment</td>
<td>stores textual comments</td>
</tr>
<tr>
<td>□</td>
<td>create request</td>
<td>spawns an SDL process dynamically</td>
</tr>
<tr>
<td>□</td>
<td>procedure call</td>
<td>invokes a procedure inside an SDL process</td>
</tr>
<tr>
<td>○</td>
<td>procedure start</td>
<td>denotes the start of a procedure</td>
</tr>
<tr>
<td>✗</td>
<td>procedure return</td>
<td>denotes the end of a procedure</td>
</tr>
</tbody>
</table>

The first four symbols after the *start* symbol exemplify the various possible set operations. SDL sets can also be compared using relational operators such as "and", "or", "\(\)", "\{=\}", "\{\}". SDL bags are similar to sets except they can contain identical items. The last four symbols in Figure C.6 show the two ways to access data members inside the SDL struct `aStruct` and the two possible array operations. All SDL arrays are of variable size and are unbounded.

A special data type called `timer` can be used to specify timing-related behavior. In Figure C.7, the declaration symbol instantiates a timer variable called `t`. In the first task symbol, this timer is configured to activate after 100 time units using the command `set`. If the SDL process receives no other signal before the activation of the timer, it receives a signal that has the same name as the timer. In the second task symbol, the command
reset deactivates the timer. However, even with the use of timers, SDL cannot be used for modeling performance as the time for transitions and signal delivery is undefined in the SDL standard.

Figure C.6: An SDL data type example.

Figure C.7: An SDL timer example.
Appendix D

PasswordCheck Models

In the formal modeling method evaluation, discussed in Section 4.4, we modeled a simplified version of PasswordCheck using three industrial methods: SDT [Tel98] from Telelogic, TestMaster [Ter98] from Teradyne, and Validator/Req [Aon98] from Aonix. The resulting models are shown in this Appendix. An overview of PasswordCheck can be found in Section 3.3.1.

D.1 SDL Model/MSCs

Figure D.3 shows the first page of the 3-page SDL [CCI88] specification that we created for the simplified PasswordCheck using Telelogic SDT. Readers are encouraged to consult Appendix C for more details on SDL and Section 4.6.1 for information on MSCs [IT96] and the tool SDT. The SDL process shown in this diagram models the behavior of PasswordCheck when touch-tone keys are received. On the top-left corner of the Figure, $maxRetry$ and $password$ are the two parameters to the SDL process for representing persistent data of the component: the maximum number of invalid password attempts and the numerical value of the predefined password. The explanations for local variables used in the process can be found in the the declaration symbol near the top-right corner of the diagram (see Figure D.1 for a list of commonly used SDL symbols).

The process begins at the Start Symbol and outputs the signal $initialPrompt$ to the environment to denote the action of playing the initial prompt. After activating the timer $noRespT$ for detecting a possible idle timeout, the process waits for inputs from the caller, which can be the * key, the # key or a digit key. In the first case, the timer is deactivated, the signal $helpPrompt$ is sent, and the password entry is clear. The scenario
is similar when a digit key is received; however, the received digit number is accumulated in the variable numberRecv, and the variable isDigitEntered is modified to indicate that a partial password has been entered. When the key # is detected, this variable determines whether the process should attempt a password match or exit prematurely. In the cases where the caller enters a correct password or requests a premature exit, the process sends a parameterized internal signal done to indicate the reason of termination. When a timeout occurs, the flow of control is transferred to the connector processTimeout. The processing of this event is not shown in the diagram.

Figure D.4 shows an MSC that describes one particular scenario between a caller and PasswordCheck. These two participating objects are represented by the Start symbols named caller and pwChkInstance in the chart, respectively (refer to Figure D.2 for a list of commonly use MSC symbols). The name Process passwordCheck near the object on the right associates the object with the SDL process in Figure D.3.

This MSC describes the case where a caller hears an initial prompt and then enters an incorrect password “8”. After staying idle and hearing a delay prompt, she eventually enters a correct password “833”. These interactions are represented by Message symbols with names that match those of the signals in Figure D.3. Some of the interactions, such as the message digit, are parameterized with the actual parameters shown in square brackets. We used a timer symbol to describe the time-out behavior, and the duration was arbitrary chosen as 100 time units. A total of 15 MSCs were created for PasswordCheck.
Figure D.3: The first page of the SDL model created for the simplified PasswordCheck.
Figure D.4: An MSC created for the simplified PasswordCheck.
D.2 TestMaster Model

Figure D.5 shows the TestMaster model that we built for the simplified PasswordCheck. Readers are encouraged to consult Section 4.6.2 for more information on TestMaster and its notation. The Figure shows the top-level finite state machine (FSM) that begins at the Entry State symbol (see Figure D.6 for a list of commonly used symbols). The two outgoing arrows denote the two mutually exclusive transitions. We describe these transitions only by their action and test information fields. In the first field, the variable maxRetry is modified to reflect the two possible settings for the maximum invalid password attempts. We put text fragments used for generating test cases in the second field.

![Figure D.5: The first-level FSM of a TestMaster model.](image)

Figure D.5: The first-level FSM of a TestMaster model.

![Figure D.6: Commonly used symbols in the TestMaster language.](image)

Figure D.6: Commonly used symbols in the TestMaster language.

After hearing the initial prompt, a tester is expected to enter a variety of touch-tone keys to test the different system behaviors. The set of possible key combinations is explicitly defined in the enterPW table, show in Figure D.7. This table lists the different key combinations in column 2 and their interpretations in column 3. During the test case generation, one row is picked from the table for each test case, and the variables passwordValue and passwordType store the values in columns 2 and 3, respectively. We chose to abstract the input as a string of touch-tone keys, rather than each individual
After selecting a password, the *validatePW* symbol is activated and transfers the control flow to the second-level FSM shown in Figure D.8. The five symbols in the middle of the Figure represent the possible system responses to different password types. For instance, if the tester enters an invalid password, the transition that leads to *invalidPrompt* symbol is activated. Figure D.9 shows a screen shot of the property editor with details of this transition. Normally, properties are shown in full detail together the FSM model, such as those in Figure D.5. However, circled T’s are used if there is not enough screen space.

In Figure D.9, the expression in the *predicate* field denotes that the transition is enabled only when the variable *passwordType* contains the value “invalid”, which implies that the variable *passwordValue* have the value “2951#” (they belong to the same row in Figure D.7). During the test case generation, the word “passwordValue” in the *test information* field is replaced by the actual variable content. Transitions that handle other password types are similar. Afterwards, depending on the situations such as whether the maximum invalid retry limit is reached or whether a help prompt is invoked, the FSM exits at the symbol *Exit* or *Retry* and returns to the top-level FSM. Then, the transition with a constraint that have the same name of the exiting state is taken. There are five FSMs and one *Table* symbol in the entire TestMaster model, amounting to approximately six pages of specifications.

Figure D.10 shows two test scripts that were generated from the PasswordCheck model. Readers may notice that the actual content inside the variable *passwordValue* is used in line 3 of Scenario 1 and lines 3 and 5 of Scenario 2. Depending on the desirable test
coverage, the number of test cases that can be generated from the model varies, ranging from 13 for minimum transition coverage to 8922 for full unconstrained coverage.
Scenario 1
Set the maximum invalid retry to three.
Verify that the application plays the default initial prompt.
Enter a valid password: 2951#.
Verify that the application exits via 'Password 1' output.
End Scenario

Scenario 2
Set the maximum invalid retry to one.
Verify that the application plays the default initial prompt.
Enter an invalid password: 2951#.
Verify that the application plays invalid password prompt.
Enter an invalid password: 2951#.
Verify that the application exits via the 'Max. Invalid' output.
End Scenario

Figure D.9: A description of a transition in a TestMaster model.

Figure D.10: Two TestMaster test scripts generated from the PasswordCheck model.
D.3 Use Case Diagrams/Sequence Diagrams

Figure D.11 shows the use case diagram that we created for the simplified PasswordCheck using Aonix Validator/Req. Readers are encouraged to consult Section 4.6.3 for information on Validator/Req and the notation used. The two actors in the diagram, caller and administrator, denote two possible types of users that interact with the component and are shown using "stick men". The rectangle titled passwordCheck defines the system boundary of the component. This system provides two services: letting an administrator setup the component and validating passwords entered by a caller. The use cases validatePassword and componentSetup, represented by the two ellipses inside the rectangle, denote these two services, respectively. A line that connects an actor and a use case signifies a "use" relationship, e.g., a caller uses the digit-validation feature of PasswordCheck. Technically, the purpose of specifying a use case diagram in Validator/Req is only to identify the participating objects in the sequence diagrams of each use case. Since we considered only the run-time behavior of PasswordCheck in our evaluation, we concentrated our effort in refining the use case validatePassword.

![Figure D.11: The use case diagram for the simplified PasswordCheck.](image)

Figures D.12 to D.14 show three of the 20 sequence diagrams that we created for this use case. The first Figure shows the interactions between a caller and the system when the caller enters an invalid password. In the title found at the top of the Figure, the invalidPW part is the name of the sequence diagram, and the UmlUseCase:validatePassword part signifies that it is a diagram within the use case validatePassword. The actor caller and the system passwordCheck are the two participating objects, and their instances are represented by the two vertical lines below their names. The arrows pointing to and coming out from these lines are the synchronous messages sent between the actor and
the system. For instance, \textit{enterNumber} is the first message that the caller sends to the system, and its parameter denotes an invalid password that the caller enters. Assuming that the valid password is chosen to be “1234”, the parameter \textit{invalidPWType} can be defined as an integer data type that has a maximum value of “9999”, a minimum value of “1235”, and a resolution value of “1” (i.e., a domain that does not include “1234”). While it is inappropriate to choose these arbitrary boundaries for the parameter as the original specification never specifies such restrictions, these boundaries are necessary for Validator/Req to generate test cases from this sequence diagram.

Figure D.12: A sequence diagram describing the interactions that take place when a caller enters an invalid password.

Figure D.13 shows a similar sequence diagram that illustrates the case where a caller enters a valid password. The major difference is that the message \textit{enterNumber} uses a user-defined integer data type \textit{validPWType} that carries only the value “1234”. Figure D.14 shows the scenario where a caller stays idle and the system plays a delay prompt. However, since accurate timing requirements cannot be represented in Validator/Req’s sequence diagrams, we simulated the time-out behavior by having the system send a self-addressed message.

Figure D.13: A sequence diagram describing the interactions that take place when a caller enters a valid password.

Table D.1 shows four test cases that were obtained by running the Validator/Req test